



C语言从入门到精通

11小时语音视频讲解

明日科技 编著

☑实例资源库 ☑模块资源库 ☑项目资源库
☑面试资源库 ☑测试题库系统 ☑PPT电子课件

（第4版）

循序渐进，实战讲述

基础知识➪核心技术➪高级应用➪项目实战

168个应用实例，32个实践练习，1个项目案例

海量资源，可查可练

除本书配套的11小时视频讲解外，根据学习顺序，资源包还额外配备如下海量开发资源库：

实例资源库（881个实例）➪模块资源库（15个典型模块）➪项目资源库（15个项目案例）➪测试题库系统（616道测试题）➪面试资源库（371个面试真题）



在线解答，高效学习



清华大学出版社

软件开发视频大讲堂

C 语言从入门到精通

（第 4 版）

明日科技　编著

清华大学出版社

北　京

内 容 简 介

《C语言从入门到精通（第4版）》从初学者的角度出发，以通俗易懂的语言，丰富多彩的实例，详细介绍了使用C语言进行程序开发需要掌握的各方面知识。全书共分为17章，包括C语言概述、算法、数据类型、运算符与表达式、常用的数据输入/输出函数、选择结构程序设计、循环控制、数组、函数、指针、结构体和共用体、位运算、预处理、文件、存储管理、网络套接字编程和学生成绩管理系统。书中所有知识都结合具体实例进行介绍，涉及的程序代码给出了详细的注释，读者可以轻松领会C语言程序开发的精髓，快速提高开发技能。

另外，本书除了纸质内容之外，附赠资源包中还给出了海量开发资源库，主要内容如下：

☑ 语音视频讲解：总时长11小时，共126段
☑ 实例资源库：881个实例及源码详细分析
☑ 模块资源库：15个经典模块开发过程完整展现
☑ 项目案例资源库：15个企业项目开发过程完整展现
☑ 测试题库系统：616道能力测试题目
☑ 面试资源库：371个企业面试真题
☑ PPT电子教案

本书可作为软件开发入门者的自学用书，也可作为高等院校相关专业的教学参考书，还可供开发人员查阅、参考。

图书在版编目（CIP）数据

C语言从入门到精通/明日科技编著．—4版．—北京：清华大学出版社，2019
（软件开发视频大讲堂）
ISBN 978-7-302-52146-4

Ⅰ．①C…　Ⅱ．①明…　Ⅲ．①C语言-程序设计　Ⅳ．①TP312.8

中国版本图书馆CIP数据核字（2019）第009042号

责任编辑：贾小红
封面设计：刘　超
版式设计：魏　远
责任校对：马子杰
责任印制：杨　艳

出版发行：清华大学出版社
　　网　　址：http://www.tup.com.cn，http://www.wqbook.com
　　地　　址：北京清华大学学研大厦A座　　**邮　　编：**100084
　　社 总 机：010-62770175　　**邮　　购：**010-62786544
　　投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
　　质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市金元印装有限公司
经　　销：全国新华书店
开　　本：203mm×260mm　　**印　　张：**26.25　　**字　　数：**721千字
版　　次：2010年7月第1版　2019年3月第4版　　**印　　次：**2019年3月第1次印刷
定　　价：69.80元

产品编号：080593-01

如何使用本书开发资源库

在学习《C 语言从入门到精通（第 4 版）》时，随书附赠的资源包中提供了“Visual C++开发资源库”系统，可以帮助读者快速提升编程水平和解决实际问题的能力。《C 语言从入门到精通（第 4 版）》和“Visual C++开发资源库”配合学习的流程如图 1 所示。

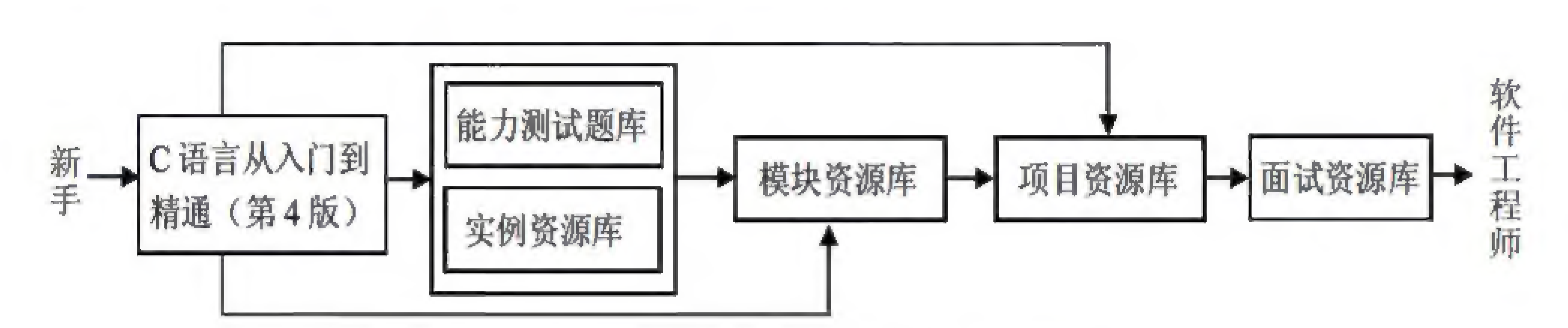


图 1　本书与附赠开发资源库配合学习的流程

打开资源包中的“开发资源库”文件夹，运行 Visual C++开发资源库.exe 程序，即可进入“Visual C++开发资源库”系统，主界面如图 2 所示。

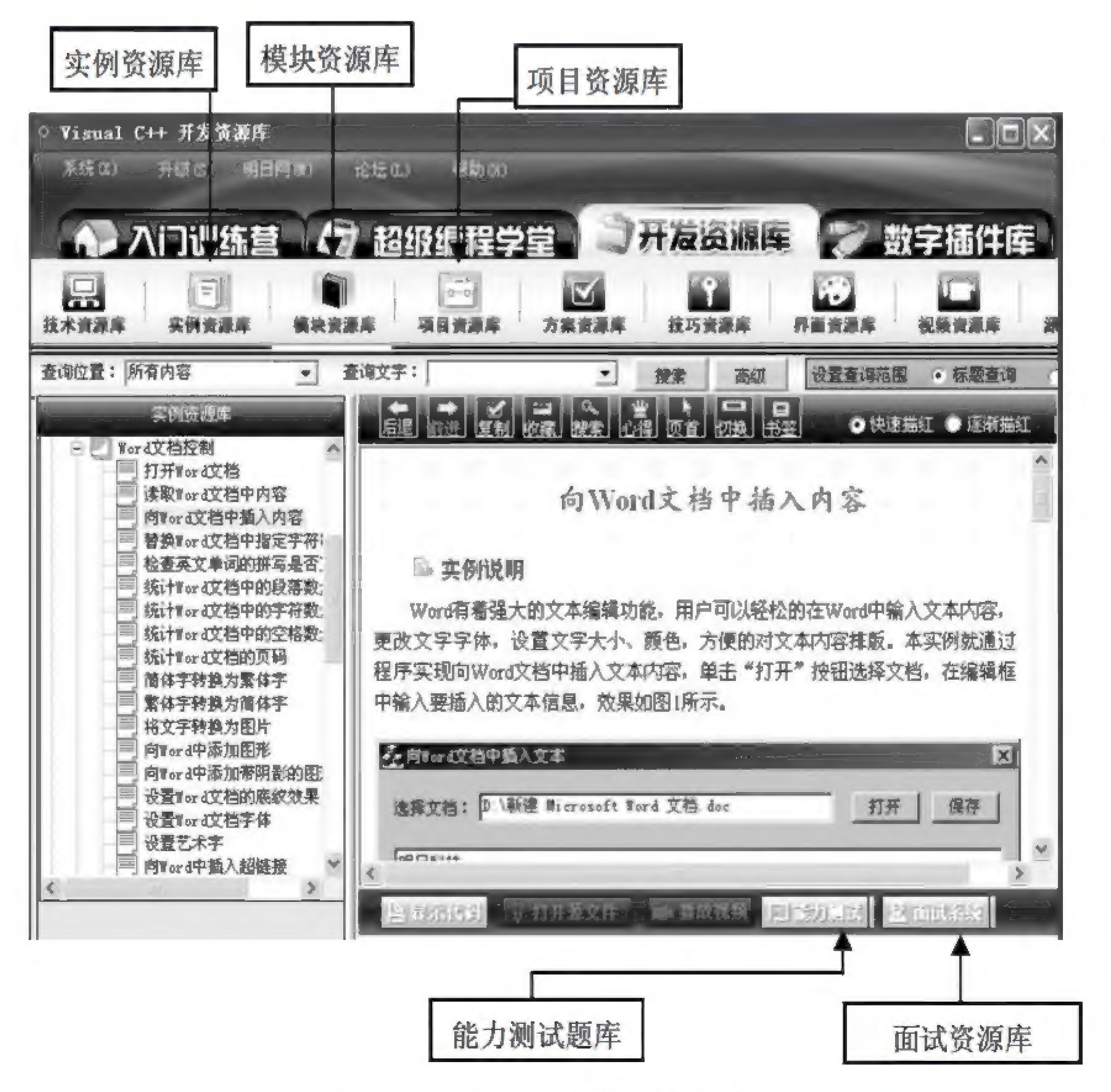


图 2　Visual C++开发资源库主界面

在学习本书某一章节时，可利用实例资源库对应章节提供的大量热点实例和关键实例巩固所学编程技能，提高编程兴趣和自信心；也可以利用能力测试题库的对应章节进行测试，检验学习成果。具体流程如图 3 所示。

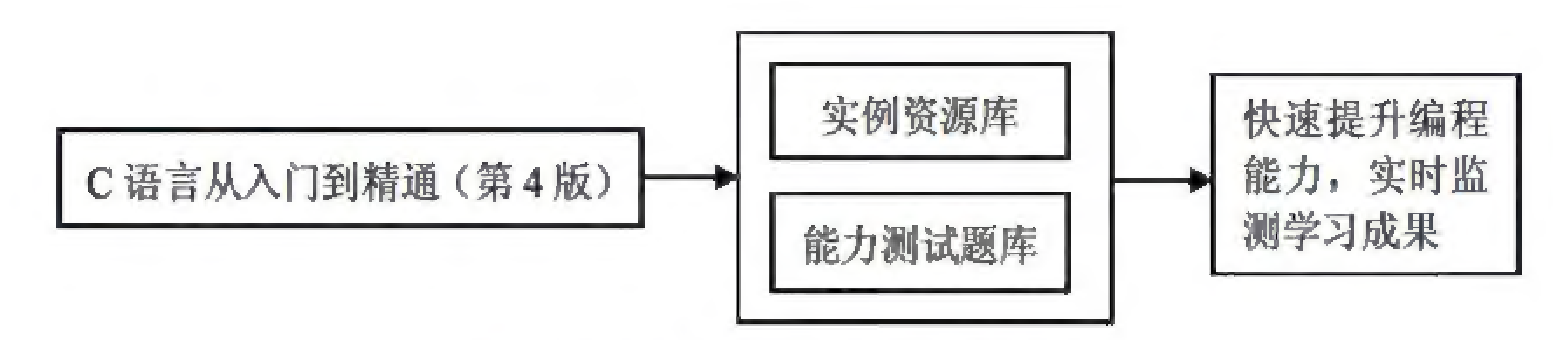


图 3　使用实例资源库和能力测试题库

对于数学逻辑能力和英语基础较为薄弱的读者，本书提供了数学及逻辑思维能力测试和编程英语能力测试，以供读者进行练习和测试，如图 4 所示。

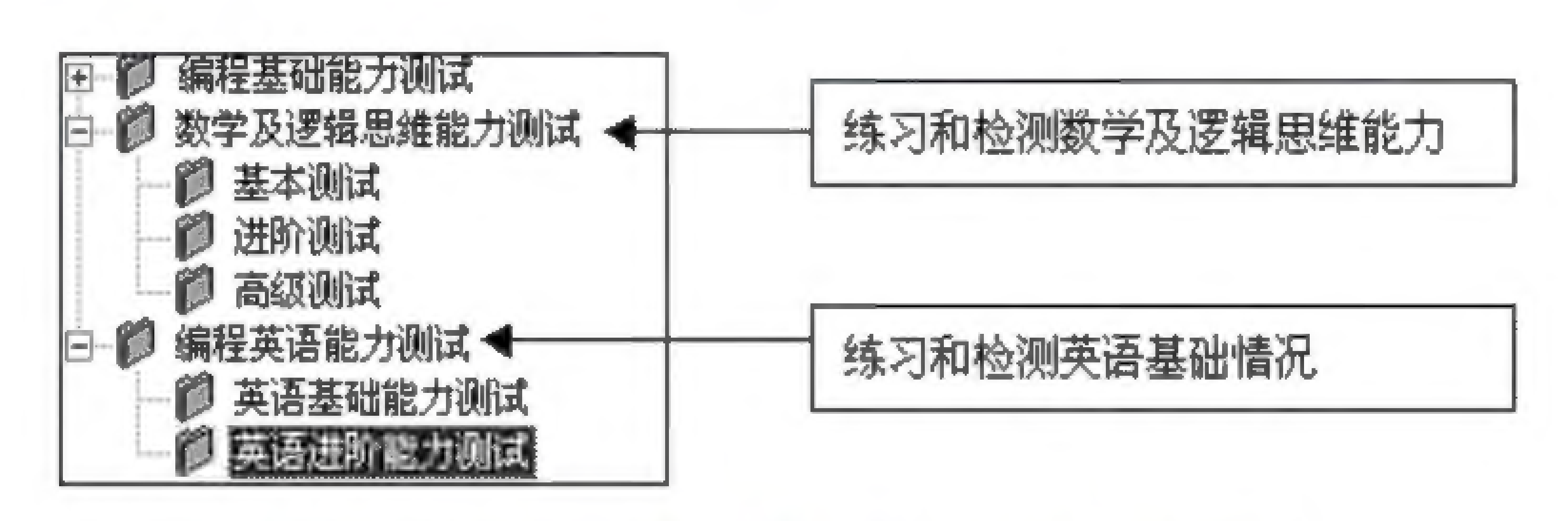


图 4　数学及逻辑思维能力测试和编程英语能力测试目录

学习完本书后，读者可通过模块资源库和项目资源库中的 30 个经典模块和项目，全面提升个人综合编程技能和解决实际开发问题的能力，为成为 C 语言软件开发工程师打下坚实基础。具体模块和项目目录如图 5 所示。

万事俱备，该到软件开发的主战场上接受洗礼了。面试资源库中提供了大量国内外软件企业的常见面试真题，同时还提供了程序员职业规划、程序员面试技巧、企业面试真题汇编和虚拟面试系统等精彩内容，是程序员求职面试的绝佳指南。面试资源库的具体内容如图 6 所示。

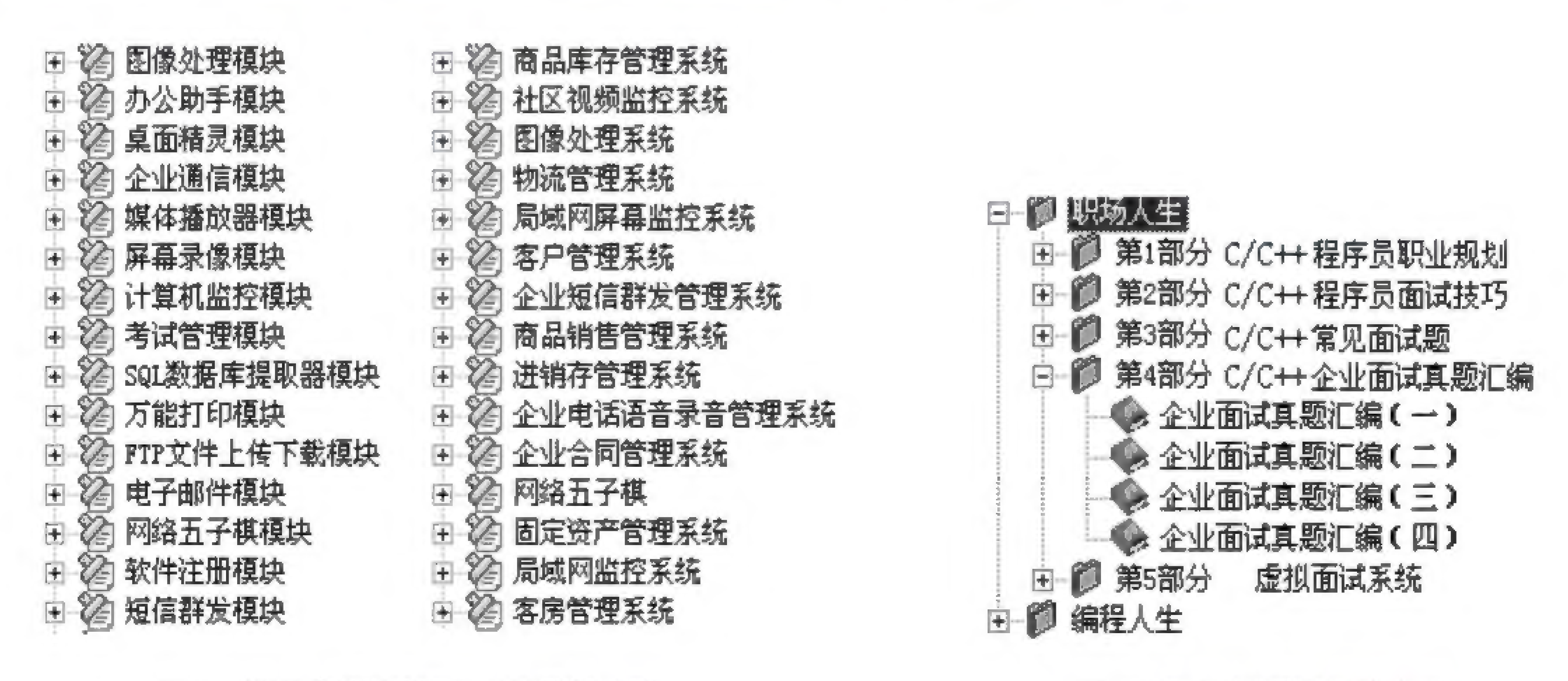
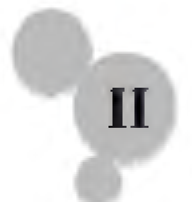


图 5　模块资源库和项目资源库目录　　　图 6　面试资源库具体内容



前　言

Preface

丛书说明：“软件开发视频大讲堂”丛书（第1版）于2008年8月出版，因其编写细腻，易学实用，配备海量学习资源和全程视频等，在软件开发类图书市场上产生了很大反响，绝大部分品种在全国软件开发零售图书排行榜中名列前茅，2009年多个品种被评为“全国优秀畅销书”。

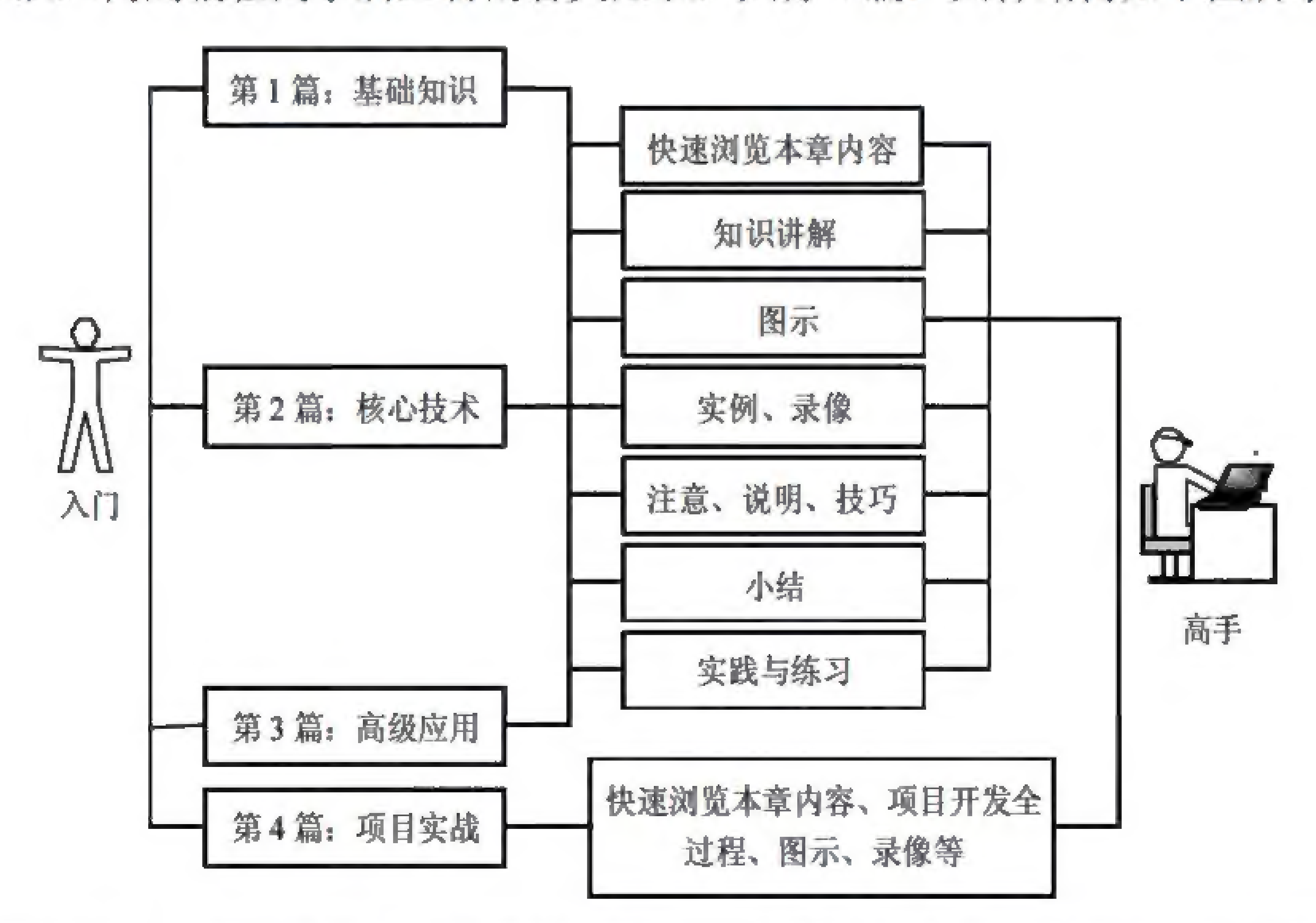
“软件开发视频大讲堂”丛书（第2版）于2010年8月出版，第3版于2012年8月出版，第4版于2016年10月出版。十年锤炼，打造经典。丛书迄今累计重印426次，销售200多万册。不仅深受广大程序员的喜爱，还被百余所高校选为计算机、软件等相关专业的教学参考用书。

“软件开发视频大讲堂”丛书（第5版）在继承前4版所有优点的基础上，将开发环境和工具全部更新为最新的JDK10和Eclipse最新版本，并且全部重新录制了视频，结合目前市场需要，进一步对丛书品种进行了完善，对相关内容进行了更新优化，使之更适合读者学习，为了方便教学，还提供了教学课件PPT。

C语言是Combined Language（组合语言）的简称。作为一种计算机设计语言，它同时具有高级语言和汇编语言两者的特点，因此受到广大编程人员的喜爱。C语言的应用非常广泛，既可以编写系统程序，也可以编写应用程序，还可以应用到单片机及嵌入式系统的开发中。这就是为什么大多数开发人员初学编程都选择C语言的原因。

本书内容

本书提供了从入门到编程高手所必备的各类知识，共分4篇，大体结构如下图所示。



第1篇：基础知识。本篇讲解了C语言基础知识，只有具备扎实的基础知识才能更快地掌握高级

的技术内容。通过对 C 语言的历史和特性、C 语言的开发环境、算法、数据类型、运算符与表达式、常用的数据输入/输出函数、选择结构程序设计和循环控制等内容的介绍，结合流程图和实例，并通过视频的指导讲解，可帮助读者为以后编程奠定坚实的基础。

第 2 篇：核心技术。本篇介绍了 C 语言的数组、函数和指针这三大部分内容，并将前面所学的基础内容融入其中，是更高级的程序设计内容。读者学习完这一部分，能够编写一些简单的 C 语言应用程序。

第 3 篇：高级应用。本篇介绍了结构体和共用体、位运算、预处理、文件、存储管理和网络套接字编程的内容。读者学习完这一部分，能够设计出较复杂的程序，并且涉及的范围更广。

第 4 篇：项目实战。本篇通过一个大型的学生成绩管理系统，运用软件工程的设计思想，介绍如何进行软件项目的开发。书中按照“需求分析→系统设计→功能设计”的步骤，带领读者一步一步地亲身体验项目开发的全过程。

本书特点

☑ **由浅入深，循序渐进。**本书以初、中级程序员为对象，先从 C 语言基础学起，再学习 C 语言的程序结构，然后学习 C 语言的高级应用，最后学习开发一个完整项目。讲解详尽，层次清晰，并且在叙述过程中会给出相应的实例以便于读者理解所讲解的知识。在讲解实例时分步骤进行，使读者在阅读时一目了然，从而快速把握书中内容。

☑ **微课视频，讲解详尽。**为便于读者直观感受程序开发的全过程，书中大部分章节都配备了教学微视频，使用手机扫描正文小节标题一侧的二维码，即可观看学习，能快速引导初学者入门，感受编程的快乐和成就感，进一步增强学习的信心。

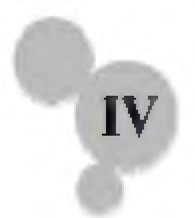
☑ **实例典型，轻松易学。**通过实例学习是最好的学习方式，本书通过“一个知识点、一个例子、一个结果、一段评析、一个综合应用”的模式，透彻、详尽地讲述了实际开发中所需的各类知识。另外，为了便于读者阅读程序代码，快速学习编程技能，书中几乎为每行关键代码都提供了注释。

☑ **精彩栏目，贴心提醒。**本书根据需要在各章安排了很多“注意”“说明”“技巧”等小栏目，读者可以在学习过程中更轻松地理解相关知识点及概念，更快地掌握个别技术的应用技巧。

☑ **应用实践，随时练习。**书中几乎每章都提供了“实践与练习”，读者通过对问题的解答可重新回顾、熟悉所学的知识，举一反三，为进一步学习做好充分的准备。

读者对象

☑ 初学编程的自学者
☑ 编程爱好者
☑ 大中专院校的老师和学生
☑ 相关培训机构的老师和学员
☑ 做毕业设计的学生
☑ 初、中级程序开发人员
☑ 程序测试及维护人员
☑ 参加实习的“菜鸟”级程序员



读者服务

学习本书时，请先扫描封底的权限二维码（需要刮开涂层）获取学习权限，然后即可免费学习书中的所有线上线下资源。本书所附赠的各类学习资源，读者可登录清华大学出版社网站（www.tup.com.cn），在对应图书页面下获取其下载方式。也可扫描图书封底的“文泉云盘”二维码，获取其下载方式。

为了方便解决本书疑难问题，读者朋友可加我们的企业 QQ：4006751066（可容纳 10 万人），也可以登录 www.mingrisoft.com 留言，我们将竭诚为您服务。

致读者

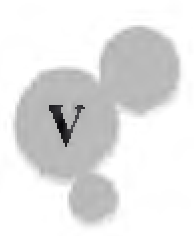
本书由明日科技 C 语言程序开发团队组织编写，主要参与编写的人员有李菁菁、王小科、赛奎春、周佳星、王国辉、李磊、贾景波、张鑫、赵宁、申小琦、冯春龙、白宏健、何平、申野、赵宁、王赫男、辛洪郁、宋万勇、潘建羽、隋妍妍、卞昉、葛忠月、张渤洋、乔宇、杨柳、林驰、岳彩龙、李春林、李雪、李颖、朱艳红、杨丽、高春艳、张宝华、张云凯、庞凤、白兆松、依莹莹、王欢、梁英、刘媛媛、胡冬、宋禹蒙等。在编写过程中，我们以科学、严谨的态度，力求精益求精，但错误、疏漏之处在所难免，敬请广大读者批评指正。

感谢您购买本书，希望本书能成为您编程路上的领航者。

“零门槛”编程，一切皆有可能。

祝读书快乐！

编　者



目　录

Contents

第1篇　基 础 知 识

第2篇 核心技术



第3篇　高级应用

第 4 篇　项 目 实 战

资源包“开发资源库”目录

第 1 大部分　实例资源库

（881 个完整实例分析，资源包路径：开发资源库/实例资源库）

第 2 大部分　模块资源库

（15 个经典模块，资源包路径：开发资源库/模块资源库）

第 3 大部分 项目资源库

（15 个企业开发项目，资源包路径：开发资源库/项目资源库）

第 4 大部分　能力测试题库

（616 道能力测试题目，资源包路径：开发资源库/能力测试）

第 1 部分　Visual C++ 编程基础能力测试

……

第 2 部分　数学及逻辑思维能力测试

- 基本测试
- 进阶测试
- 高级测试

第 3 部分　编程英语能力测试

- 英语基础能力测试
- 英语进阶能力测试

第 5 大部分　面试资源库

（371 项面试真题，资源包路径：开发资源库/面试系统）

第 1 部分　C、C++程序员职业规划

- 你了解程序员吗
- 程序员自我定位

第 2 部分　C、C++程序员面试技巧

- 面试的 3 种方式
- 如何应对企业面试
- 英语面试
- 电话面试
- 智力测试

第 3 部分　C、C++常见面试题

- C/C++语言基础面试真题
- 字符串与数组面试真题
- 函数面试真题
- 指针与引用面试真题
- 预处理和内存管理面试真题
- 位运算面试真题
- 面向对象面试真题
- 继承与多态面试真题
- 数据结构与常用算法面试真题
- 排序与常用算法面试真题

第 4 部分　C、C++企业面试真题汇编

- 企业面试真题汇编（一）
- 企业面试真题汇编（二）
- 企业面试真题汇编（三）
- 企业面试真题汇编（四）

第 5 部分　VC 虚拟面试系统

……



基础知识

只有具备扎实的基础知识，才能更快地掌握高级技术。本篇讲解了 C 语言的基础知识，包括 C 语言的历史和特性、C 语言的开发环境、算法、C 语言的数据类型、运算符与表达式、常用的数据输入/输出函数、选择结构程序设计和循环控制等。在学习过程中结合流程图和实例，并通过观看教学视频，读者可为今后的编程奠定坚实的基础。

第 1 章

C 语言概述

（视频讲解：33 分钟）

在学习 C 语言之前，首先要了解 C 语言的发展历程，这是每一个初学 C 语言的人员都应该清楚的，并且应了解为什么要选择 C 语言，以及它有哪些特性。只有了解了 C 语言的历史和特性，才会更深刻地了解这门语言，增加今后学习 C 语言的信心。随着计算机科学的不断发展，C 语言的学习环境也在不断发生着变化，C 语言刚出现时，大多数人会选择相对简单一些的编译器，如 Visual C++ 6.0。但是现在更多的人倾向于选择由 Microsoft 公司推出的 Visual Studio 编译器。

本章致力于使读者了解 Visual C++ 6.0 和 Visual Studio 2017 开发环境，掌握其中各个部分的使用方法，并通过编写简单的应用程序，以练习使用开发环境。

通过阅读本章，您可以：

- 了解 C 语言的发展史
- 了解 C 语言的特点
- 了解 C 语言的组织结构
- 掌握如何使用 Visual C++ 6.0 开发 C 程序
- 掌握如何使用 Visual Studio 2017 开发 C 程序

1.1　C 语言的发展史



1.1.1　程序语言简述

在介绍 C 语言的发展历程之前，应先对程序语言有一个大概的了解。

1. 机器语言

机器语言是低级语言，也称为二进制代码语言。计算机使用的是由 0 和 1 组成的二进制数组成的一串指令来表达计算机操作的语言。机器语言的特点是，计算机可以直接识别，不需要进行任何的翻译。

2. 汇编语言

汇编语言是面向机器的程序设计语言。为了减轻使用机器语言编程的痛苦，用英文字母或符号串来替代机器语言的二进制码，这样就把不易理解和使用的机器语言变成了汇编语言。因此，汇编语言要比机器语言更便于阅读和理解。

3. 高级语言

由于汇编语言依赖于硬件体系，并且该语言中的助记符号数量比较多，所以其运用起来仍然不够方便。为了使程序语言能更贴近人类的自然语言，同时又不依赖于计算机硬件，于是产生了高级语言。这种语言，其语法形式类似于英文，并且因为不需要对硬件进行直接操作，因此易于被普通人所理解与使用。其中影响较大、使用普遍的高级语言有 Fortran、ALGOL、Basic、COBOL、LISP、Pascal、PROLOG、C、C++、VC、VB、Delphi、Java 等。

1.1.2　C 语言的历史

从程序语言的发展过程可以看到，以前的操作系统等系统软件主要是用汇编语言编写的。但由于汇编语言依赖于计算机硬件，程序的可读性和可移植性都不是很好，为了提高可读性和可移植性，人们开始寻找一种语言，这种语言应该既具有高级语言的特性，又不失低级语言的优点。于是，C 语言产生了。

C 语言是在 BCPL 语言（简称 B 语言）的基础上发展和完善起来的，而 B 语言是由 UNIX 的研制者丹尼斯 • 里奇（Dennis Ritchie）和肯 • 汤普逊（Ken Thompson）于 1970 年研制出来的。20 世纪 70 年代初期，AT&T Bell 实验室的程序员丹尼斯 • 里奇第一次把 B 语言改为 C 语言。

最初，C 语言运行于 AT&T 的多用户、多任务的 UNIX 操作系统上。后来，丹尼斯 • 里奇用 C 语言改写了 UNIX C 的编译程序，UNIX 操作系统的开发者肯 • 汤普逊又用 C 语言成功地改写了 UNIX，从此开创了编程史上的新篇章。UNIX 成为第一个不是用汇编语言编写的主流操作系统。

1983 年，美国国家标准委员会（ANSI）对 C 语言进行了标准化，于 1983 年颁布了第一个 C 语言草案（83ANSI C），后来于 1987 年又颁布了另一个 C 语言标准草案（87ANSI C），最新的 C 语言标准 C99 于 1999 年颁布，并在 2000 年 3 月被 ANSI 采用。但是由于未得到主流编译器厂家的支持，C99 并未得到广泛使用。

尽管 C 语言是在大型商业机构和学术界的研究实验室中研发的，但是当开发者们为第一台个人计算机提供 C 编译系统之后，C 语言就得以广泛传播，并为大多数程序员所接受。对 MS-DOS 操作系统来说，系统软件和实用程序都是用 C 语言编写的。Windows 操作系统大部分也是用 C 语言编写的。

C 语言是一种面向过程的语言，同时具有高级语言和汇编语言的优点。C 语言可以广泛应用于不同的操作系统，如 UNIX、MS-DOS、Microsoft Windows 及 Linux 等。

在 C 语言基础上发展起来的有支持多种程序设计风格的 C++语言、网络上广泛使用的 Java、JavaScript，以及微软的 C#语言等。也就是说，学好 C 语言之后，再学习其他语言就会比较轻松。



目前最流行的 C 语言有以下几种：

- ☑ Microsoft C 或称 MS C。
- ☑ Borland Turbo C 或称 Turbo C。
- ☑ AT&T C。



1.2　C 语言的特点

C 语言是一种通用的程序设计语言，主要用来进行系统程序设计，具有如下特点。

1. 高效性

谈到高效性，不得不说 C 语言是“鱼与熊掌”兼得。从 C 语言的发展历史也可以看到，它继承了低级语言的优点，产生了高效的代码，并具有友好的可读性和编写性。一般情况下，C 语言生成的目标代码的执行效率只比汇编程序低 10%～20%。

2. 灵活性

C 语言中的语法不拘一格，可在原有语法基础上进行创造、复合，从而给程序员更多想象和发挥的空间。

3. 功能丰富

除了 C 语言中所具有的类型之外，还可以使用丰富的运算符和自定义的结构类型来表达任何复杂的数据类型，完成所需要的功能。

4. 表达力强

C 语言的特点体现在它的语法形式与人们所使用的语言形式相似，书写形式自由，结构规范，并且只需简单的控制语句即可轻松控制程序流程，完成烦琐的程序要求。

5. 移植性好

由于 C 语言具有良好的移植性，从而使得 C 程序在不同的操作系统下，只需要简单地修改或者不

用修改即可进行跨平台的程序开发操作。

正是由于 C 语言拥有上述优点，使得它在程序员选择语言时备受青睐。



1.3　一个简单的 C 程序

在步入 C 语言程序世界之前，读者不要对 C 语言产生恐惧感，觉得这种语言应该是学者或研究人员的专利。C 语言是人类共有的财富，是普通人只要通过努力学习就可以掌握的知识。下面通过一个简单的程序来看一看 C 语言程序是什么样子。

【例 1.1】　一个简单的 C 程序。(**实例位置：资源包\TM\sl\1\1**)

本实例程序实现的功能只是显示一条信息“Hello,world！I'm coming!”，通过这个程序可以初窥 C 程序样貌。虽然这个简单的小程序只有 7 行，却充分说明了 C 程序是由什么位置开始、什么位置结束的。

```
#include<stdio.h>

int main()
{
    printf("Hello,world! I'm coming!\n");           /*输出要显示的字符串*/
    return 0;                                        /*程序返回 0*/
}
```

运行程序，显示效果如图 1.1 所示。



图 1.1　一个简单的 C 程序

现在来分析一下上面的实例程序。

1．#include 指令

实例代码中的第 1 行：

```
#include<stdio.h>
```

这个语句的功能是进行有关的预处理操作。include 称为文件包含命令，后面尖括号中的内容称为头部文件或首文件。有关预处理的内容，将会在本书第 13 章中进行详细讲解，在此读者只需先对此概念有所了解即可。

2．空行

实例代码中的第 2 行是空行。

C 语言是一个较灵活的语言，因此格式并不是固定不变、拘于一格的。也就是说，空格、空行、跳格并不会影响程序。有的读者就会问：“为什么要有这些多余的空格和空行呢？”其实这就像生活中

在纸上写字一样，虽然拿来一张白纸就可以在上面写字，但是通常还会在纸的上面印上一行一行的方格或段落，隔开每一段文字，自然就更加美观和规范。合理、恰当地使用这些空格、空行，可以使编写出来的程序更加规范，对日后的阅读和整理发挥着重要的作用。在此也提醒读者，在写程序时最好将程序书写得规范、干净。

注意

不是所有的空格都没有用，如两个关键字之间要用空格隔开（else if）。这种情况下如果将空格去掉，程序就不能通过编译。这里大家有个感性认识就好，在以后章节的学习中会慢慢领悟。

3. main 函数声明

实例代码中的第 3 行：

```
int main()
```

这一行代码代表的意思是声明 main 函数为一个返回值，是整型的函数。其中的 int 称为关键字，这个关键字代表的类型是整型。关于数据类型的内容将会在本书的第 3 章进行讲解，而函数的内容将会在本书的第 9 章进行详细介绍。

在函数中，这一部分称为函数头部分。在每个程序中都会有一个 main 函数，那么 main 函数是什么作用呢？main 函数就是一个程序的入口部分。也就是说，程序都是从 main 函数头开始执行的，然后进入 main 函数中，执行 main 函数中的内容。

4. 函数体

实例代码中的第 4～7 行：

```
{
    printf("Hello,world! I'm coming!\n");          /*输出要显示的字符串*/
    return 0;                                       /*程序返回 0*/
}
```

在上面介绍 main 函数时，提到了一个名词——函数头。读者通过这个词可以进行一下联想：既然有函数头，那也应该有函数的身体吧。没错，一个函数分为两个部分：一是函数头，一是函数体。

程序代码中的第 4 行和第 7 行这两个大括号就构成了函数体，函数体也可以称为函数的语句块。在函数体中，也就是第 5 行和第 6 行这一部分就是函数体中要执行的内容。

5. 执行语句

函数体中的第 5 行代码：

```
printf("Hello,world!I'm coming!\n");          /*输出要显示的字符串*/
```

执行语句就是函数体中要执行的动作内容。这一行代码是这个简单的例子中最复杂的。该行代码虽然看似复杂，其实也不难理解，printf 是产生格式化输出的函数，可以简单理解为向控制台进行输出文字或符号。括号中的内容称为函数的参数，在括号内可以看到输出的字符串“Hello,world!I'm

coming!”，其中还可以看到“\n”这样一个符号，称之为转义字符。转义字符的内容将会在本书的第 3 章进行介绍。

6. return 语句

函数体中的第 6 行代码：

```
return 0;
```

这行语句使 main 函数终止运行，并向操作系统返回一个整型常量 0。前面介绍 main 函数时说过返回一个整型返回值，此时 0 就是要返回的整型值。在此处可以将 return 理解成 main 函数的结束标志。

7. 代码的注释

在程序的第 5 行和第 6 行后面可以看到一段关于这行代码的文字描述：

```
printf("Hello,world! I'm coming!\n");          /*输出要显示的字符串*/
return 0;                                      /*程序返回 0*/
```

这段对代码的解释描述称为代码的注释。代码注释就是用来对代码进行解释说明，方便日后自己阅读或者他人阅读源程序时理解程序代码的含义和设计思想。其语法格式如下：

```
/*其中为注释内容*/
```


说明

虽然没有强行规定程序中一定要写注释，但是为程序代码写注释是一个良好的习惯，这会为以后查看代码带来非常大的方便。如果程序交给别人看，他人可以快速地掌握程序思想与代码作用。因此，编写规范的代码格式和添加详细的注释，是一个优秀程序员应该具备的好习惯。

1.4　一个完整的 C 程序


视频讲解

1.3 节展现了一个最简单的程序，通过 7 行代码的使用，实现了显示一行字符串的功能。通过 1.3 节的介绍，读者应该不会再对学习 C 语言发怵了。本节将在例 1.1 的基础上，对其内容进行扩充，使读者对 C 程序有一个更完整的认识。

【例 1.2】　一个完整的 C 程序。（**实例位置：资源包\TM\sl\1\2**）

本实例要实现这样的功能：有一个长方体，它的高已经给出，然后输入这个长方体的长和宽，通过输入的长、宽以及给定的高度，计算出长方体的体积。

```
#include<stdio.h>                              /*包含头文件*/
#define Height 10                              /*定义常量*/
int calculate(int Long, int Width);            /*函数声明*/
int main()                                     /*主函数 main*/
{
    int m_Long;                                /*定义整型变量，表示长度*/
```

```
    int m_Width;                          /*定义整型变量，表示宽度*/
    int result;                           /*定义整型变量，表示长方体的体积*/

    printf("长方形的高度为：%d\n",Height);   /*显示提示*/

    printf("请输入长度\n");                 /*显示提示*/
    scanf("%d",&m_Long);                  /*输入长方体的长度*/

    printf("请输入宽度\n");                 /*显示提示*/
    scanf("%d",&m_Width);                 /*输入长方体的宽度*/

    result=calculate(m_Long,m_Width);     /*调用函数，计算体积*/
    printf("长方体的体积是：");              /*显示提示*/
    printf("%d\n",result);                /*输出体积大小*/
    return 0;                             /*返回整型 0*/
}

int calculate(int Long, int Width)        /*定义计算体积函数*/
{
    int result =Long*Width*Height;        /*具体计算体积*/
    return result;                        /*将计算的体积结果返回*/
}
```

运行程序，显示效果如图 1.2 所示。



图 1.2　一个完整的 C 程序

说明

这里要再次提示一下此程序的用意。例 1.2 和例 1.1 并不是要将具体的知识点进行详细的讲解，只是将 C 语言程序的概貌显示给读者，使读者对 C 语言程序有一个简单的印象。还记得小时候学习加减法的情况吗？老师只是教给学生们“1+1=2”，却没有教给学生们“1+1 为什么等于 2”或者“如何证明 1+1=2”这样的问题。学习加减法是这样的过程，学习 C 语言编写程序也应该是这样的过程，在不断的接触中变得熟悉，在不断的思考中变得深入。

在具体讲解这个程序的执行过程之前，先展现该程序的过程图，这样可以使读者对程序有一个更为清晰的认识，如图 1.3 所示。

通过上述程序流程图，可以观察出整个程序运行的过程。前面已经介绍过程序中的一些内容，这里不再进行有关的说明。下面介绍程序中新出现的一些内容。

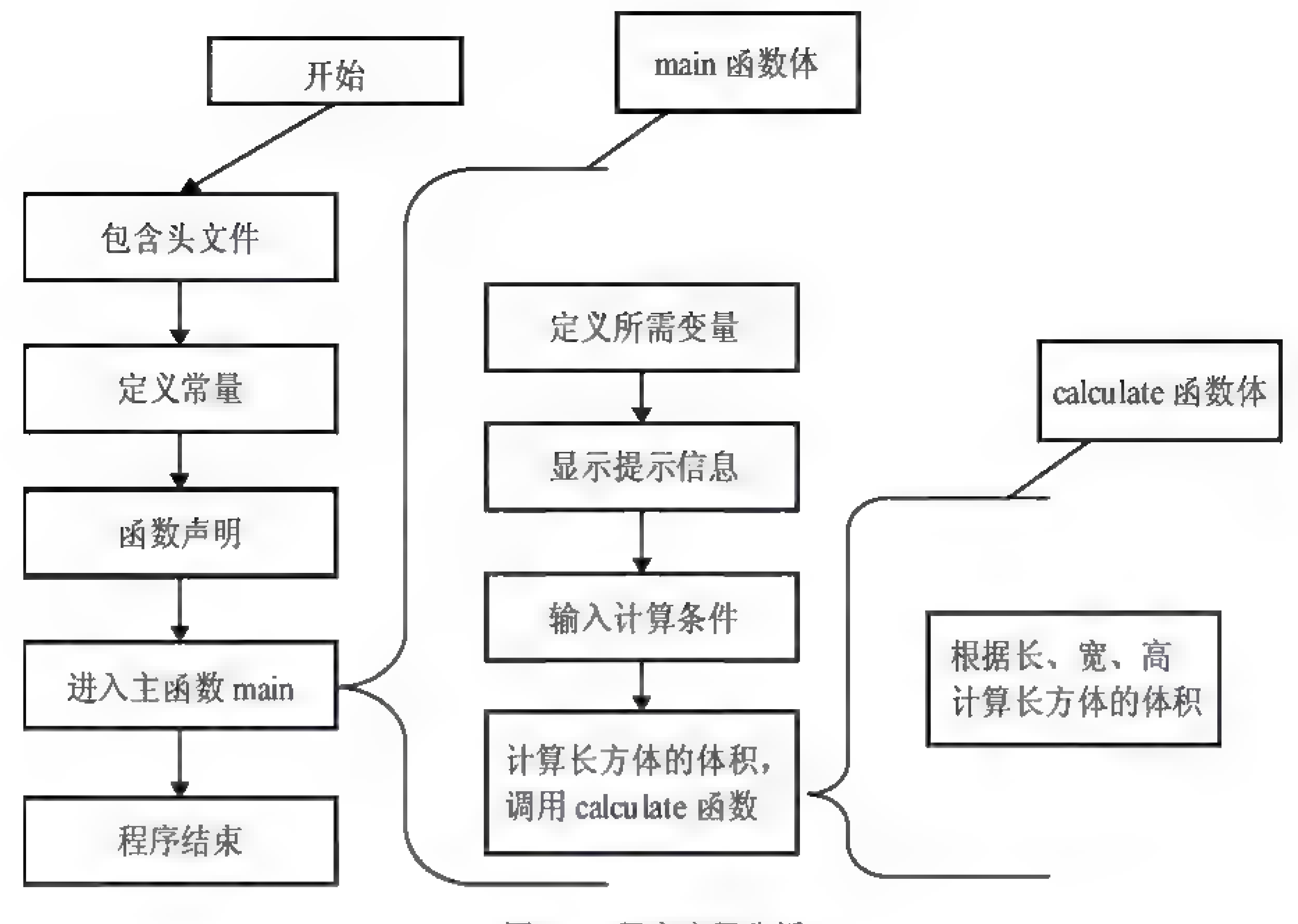


图 1.3　程序流程分析

1．定义常量

实例代码中的第 2 行：

```
#define Height 10                                /*定义常量*/
```

这一行代码中，使用#define 定义一个符号。#define 在这里的功能是设定这个符号为 Height，并且指定这个符号 Height 代表的值为 10。这样在程序中，只要是使用 Height 这个标识符的位置，就代表使用的是 10 这个数值。

2．函数声明

实例代码中的第 3 行：

```
int calculate(int Long, int Width);              /*函数声明*/
```

此处代码的作用是对一个函数进行声明。前面介绍过函数，但是什么是函数声明呢？举一个例子，两个公司进行合作，其中的 A 公司要派一个经理到 B 公司进行业务洽谈。A 公司会发送一个通知给 B 公司，告诉 B 公司会派一个经理过去，请 B 公司在机场接一下这位洽谈业务的经理。A 公司将这位经理的名字和大概的体貌特征都告诉 B 公司的有关迎接人员。这样当这位经理下飞机之后，B 公司就可以将他的名字写在纸上做成接机牌，然后找到这位经理。

声明函数的作用就像 A 公司告诉 B 公司有关这位经理信息的过程，为接下来要使用的函数做准备。也就是说，如果此处声明 calculate 函数，那么在程序代码的后面会有 calculate 函数的具体定义内容，这样程序中如果出现 calculate 函数，程序就会根据 calculate 函数的定义执行有关的操作。至于有关的具体内容将会在第 9 章进行介绍。

3. 定义变量

实例代码中的第 6～8 行：

```
int m_Long;                    /*定义整型变量，表示长度*/
int m_Width;                   /*定义整型变量，表示宽度*/
int result;                    /*定义整型变量，表示长方体的体积*/
```

这 3 行语句都是定义变量的语句。在 C 语言中要使用变量，必须在使用变量之前进行定义，之后编译器会根据变量的类型为变量分配内存空间。变量的作用就是存储数值，用变量进行计算。这就像在二元一次方程中，X 和 Y 就是变量，当为其进行赋值后，如 X 为 5，Y 为 10，这样 X+Y 的结果就等于 15。

4. 输入语句

实例代码中的第 13 行：

```
scanf("%d",&m_Long);           /*输入长方体的长度*/
```

在例 1.1 中曾经介绍过显示输出函数 printf，那么既然有输出就一定会有输入。在 C 语言中，scanf 函数就用来接收键盘输入的内容，并将输入的内容保存在相应的变量中。可以看到，在 scanf 函数的参数中，m_Long 就是之前定义的整型变量，它的作用是存储输入的信息内容。其中的“&”符号是取地址运算符，其具体内容将会在本书的后续章节中进行介绍。

5. 数学运算语句

实例代码中的第 26 行：

```
int result =Long*Width*Height;           /*具体计算体积*/
```

这行代码在 calculate 函数体内，其功能是将变量 Long、Width、Height 三者相乘得到的结果保存在 result 变量中。其中的“*”号代表乘法运算符。

以上内容已经将其中的要点知识全部提取出来，关于 C 语言程序，相信读者此时已经有了一定的了解。下面再将上面的程序执行过程进行总结：

（1）包含程序所需要的头文件。

（2）定义一个常量 Height，其代表的值为 10。

（3）对 calculate 函数进行声明。

（4）进入 main 函数，程序开始执行。

（5）在 main 函数中，首先定义 3 个整型变量，分别代表长方体的长度、宽度和体积。

（6）显示提示文字，然后根据显示的文字输入有关的数据。

（7）当长方体的长度和宽度都输入之后，会调用 calculate 函数，计算长方体的体积。

（8）定义 calculate 函数的位置在 main 函数的下面，在 calculate 函数体内将计算长方体体积的结果进行返回。

（9）在 main 函数中，result 变量得到了 calculate 函数返回的结果。

（10）通过输出语句将其中长方体的体积显示出来。

（11）程序结束。

1.5　C 语言程序的格式



通过上面两个实例的介绍，可以看出 C 语言编写有一定的格式特点：

☑　主函数 main

C 程序都是从 main 函数开始执行的。main 函数不论放在什么位置都没有关系。

☑　C 程序整体是由函数构成的

main 是程序中的主函数，当然在程序中是可以定义其他函数的。在这些定义函数中进行特殊的操作，使得函数完成特定的功能。虽然将所有的执行代码全部放入 main 函数也是可行的，但是如果将其分成一块一块，每一块使用一个函数进行表示，那么整个程序看起来就具有结构性，并且易于观察和修改。

☑　函数体的内容在"{}"中

每一个函数都要执行特定的功能，那么如何才能看出一个函数的具体操作范围呢？答案就是寻找"{"和"}"这两个大括号。C 语言使用一对大括号来表示程序的结构层次，需要注意的就是左右大括号要对应使用。

技巧

在编写程序时，为了防止对应大括号的遗漏，每次都可以先将两个对应的大括号写出来，再向括号中添加代码。

☑　每一个执行语句都以";"结尾

如果注意观察前面的两个实例，就会发现在每一个执行语句后面都会有一个";"（分号）作为语句结束的标志。

☑　英文字符大小不通用

同一大、小写字母意义不同，关键字和标准库函数名必须用小写。

☑　空格、空行的使用

前面讲解空行时已经对其进行阐述，其作用就是增加程序的可读性，使得程序代码位置合理、美观。例如，如下代码就非常不利于观察：

```
int Add(int Num1, int Num2)          /*定义计算加法函数*/
{/*将两个数相加的结果保存在 result 中*/
int result =Num1+Num2;
return result;                       /*将计算的结果返回*/}
```

如果将其中的执行语句在函数中进行缩进，使得函数体内代码开头与函数头的代码不在一列，就会很有层次感，例如：

```
int Add(int Num1, int Num2)          /*定义计算加法函数*/
{
    int result =Num1+Num2;           /*将两个数相加的结果保存在 result 中*/
```

```
    return result;                    /*将计算的结果返回*/
}
```

视频讲解

1.6 开 发 环 境

欲善工事，先利其器。要将一件事情做好，先要了解制作工具。本节将详细介绍两种学习 C 语言程序开发的常用工具，一个是 Visual C++ 6.0，另一个是 Visual Studio 2017。

1.6.1 Visual C++ 6.0

Visual C++ 6.0 是一个功能强大的可视化软件开发工具，它将程序的代码编辑、程序编译、链接和调试等功能集于一身。Visual C++ 6.0 操作和界面都比较友好，使得开发过程更快捷、方便。本书中的所有程序都是在 Visual C++ 6.0 开发环境中进行编写的。接下来将介绍 Visual C++ 6.0 的安装和使用过程。

1. Visual C++ 6.0 的安装

微软公司已经停止了对 Visual C++ 6.0 的技术支持，并且也不提供下载，本书使用的 Visual C++ 6.0 的中文版，读者可以在网上搜索，下载合适的安装包。接下来介绍安装过程。

注意

如果读者是 Win10 系统，建议安装 Visual C++ 6.0 英文版。

Visual C++6.0 的具体安装步骤如下：

（1）双击打开 Visual C++6.0 安装文件夹中的 SETUP.mp4 文件，打开的界面如图 1.4 所示，单击“运行程序”按钮进行安装。

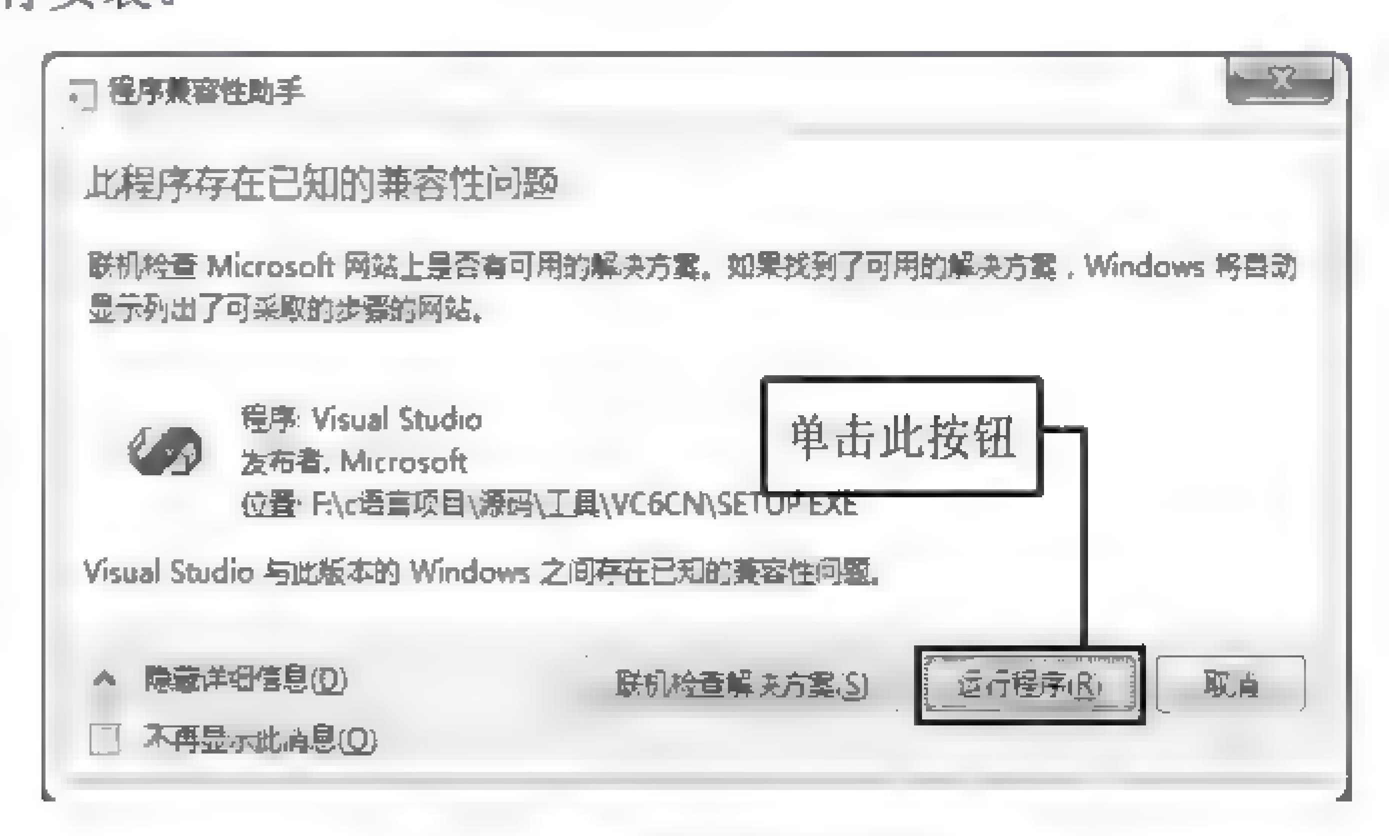


图 1.4　单击“运行程序”按钮

（2）进入“安装向导”界面，单击“下一步”按钮。进入“最终用户许可协议”界面，首先选择“接受协议”选项，然后单击“下一步”按钮。

（3）进入“产品号和用户 ID”界面，如图 1.5 所示。在安装包内找到 CDKEY.txt 文件，填写产品 ID。姓名和公司名称根据情况填写，可以采用默认设置，不对其修改，单击“下一步”按钮。

（4）进入“Visual C++ 6.0 中文企业版”界面，如图 1.6 所示，选中“安装 Visual C++ 6.0 中文企业版”单选按钮，然后单击“下一步”按钮。



图 1.5　“产品号和用户 ID”界面

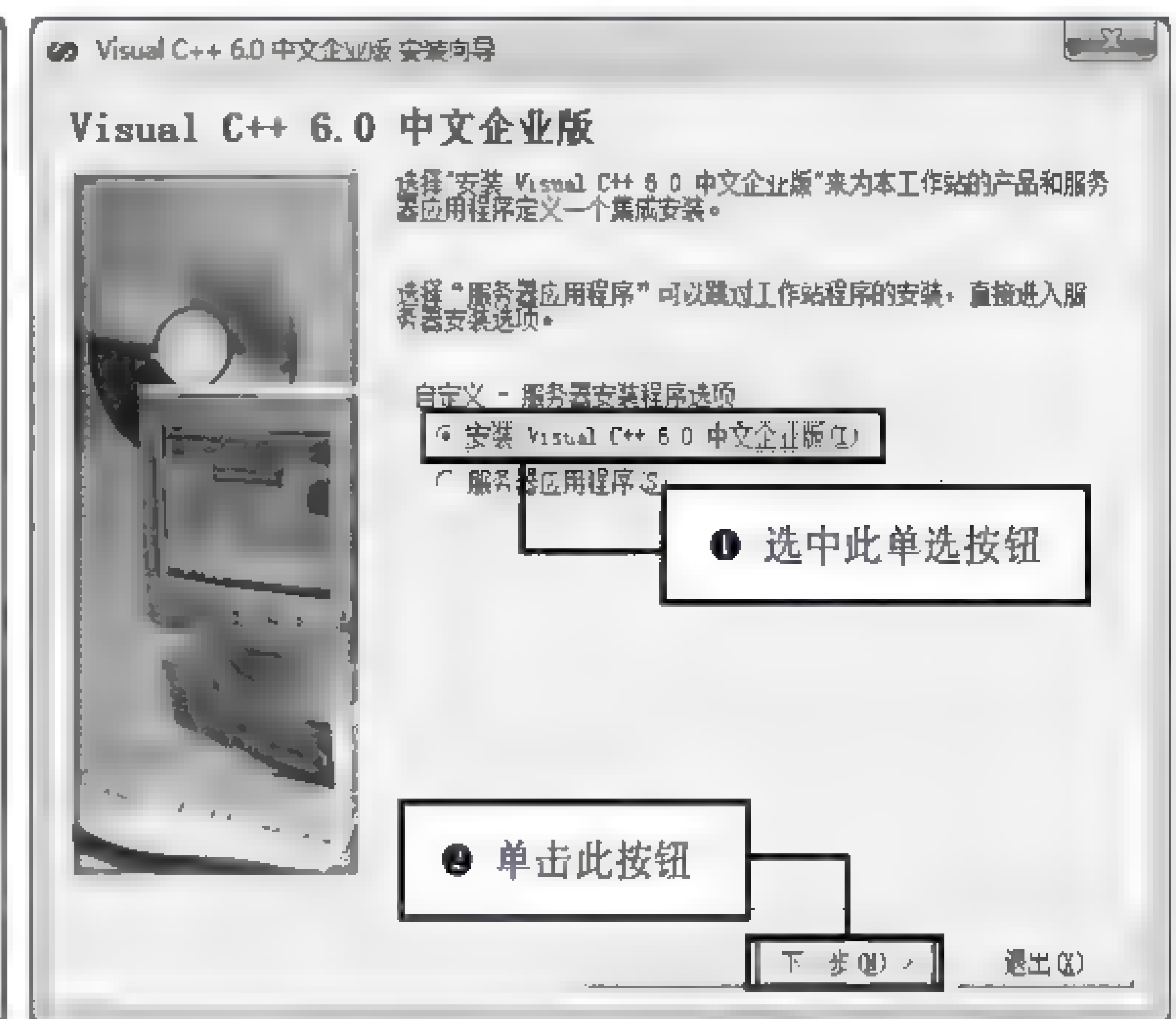


图 1.6　“Visual C++ 6.0 中文企业版”界面

（5）进入“选择公用安装文件夹”界面，如图 1.7 所示。公用文件默认是存储在 C 盘中的，单击“浏览”按钮，选择安装路径，这里建议安装在空间剩余比较大的磁盘中，单击“下一步”按钮。

（6）进入安装程序的欢迎界面，如图 1.8 所示，单击“继续”按钮。

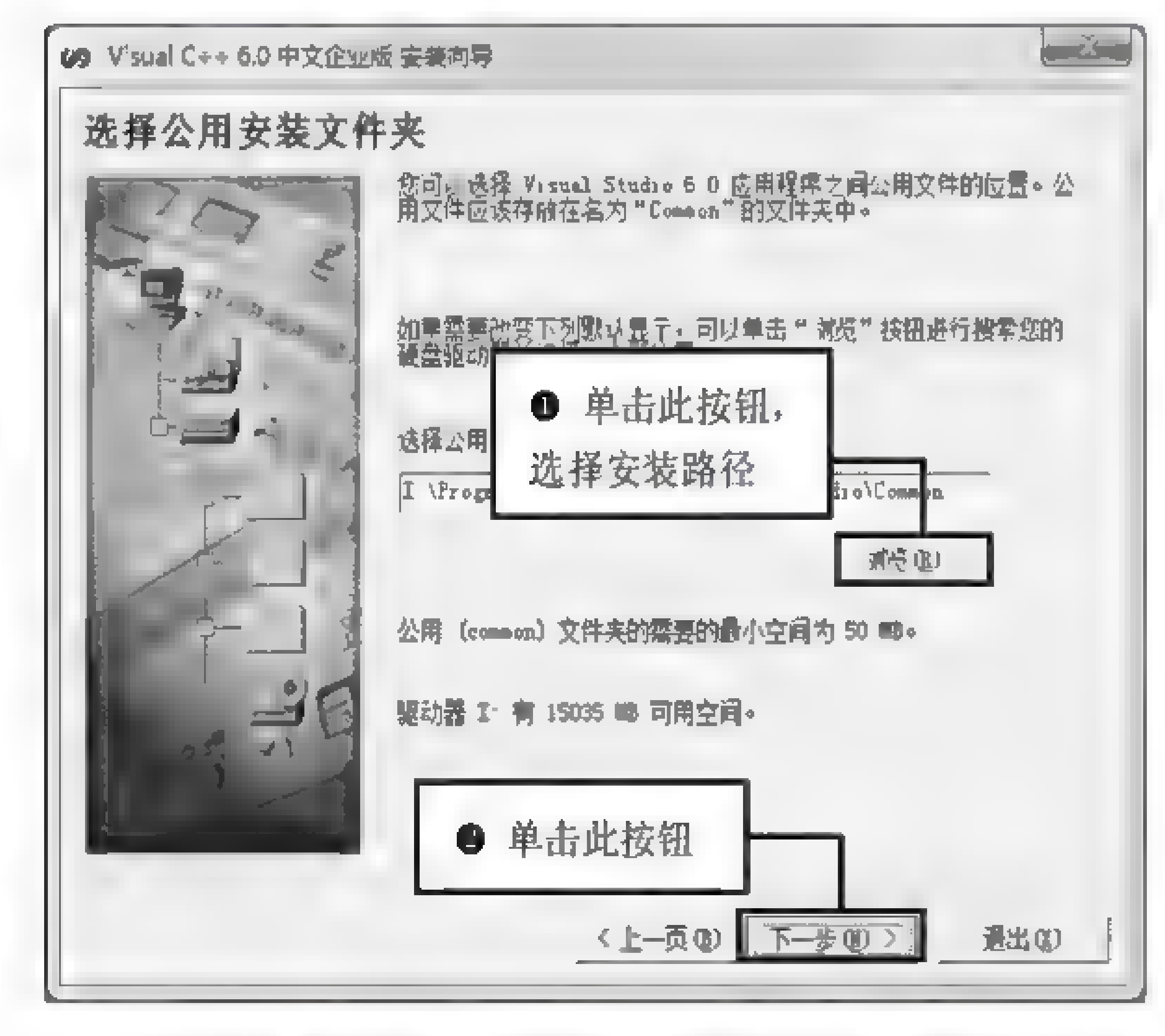


图 1.7　“选择公用安装文件夹”界面

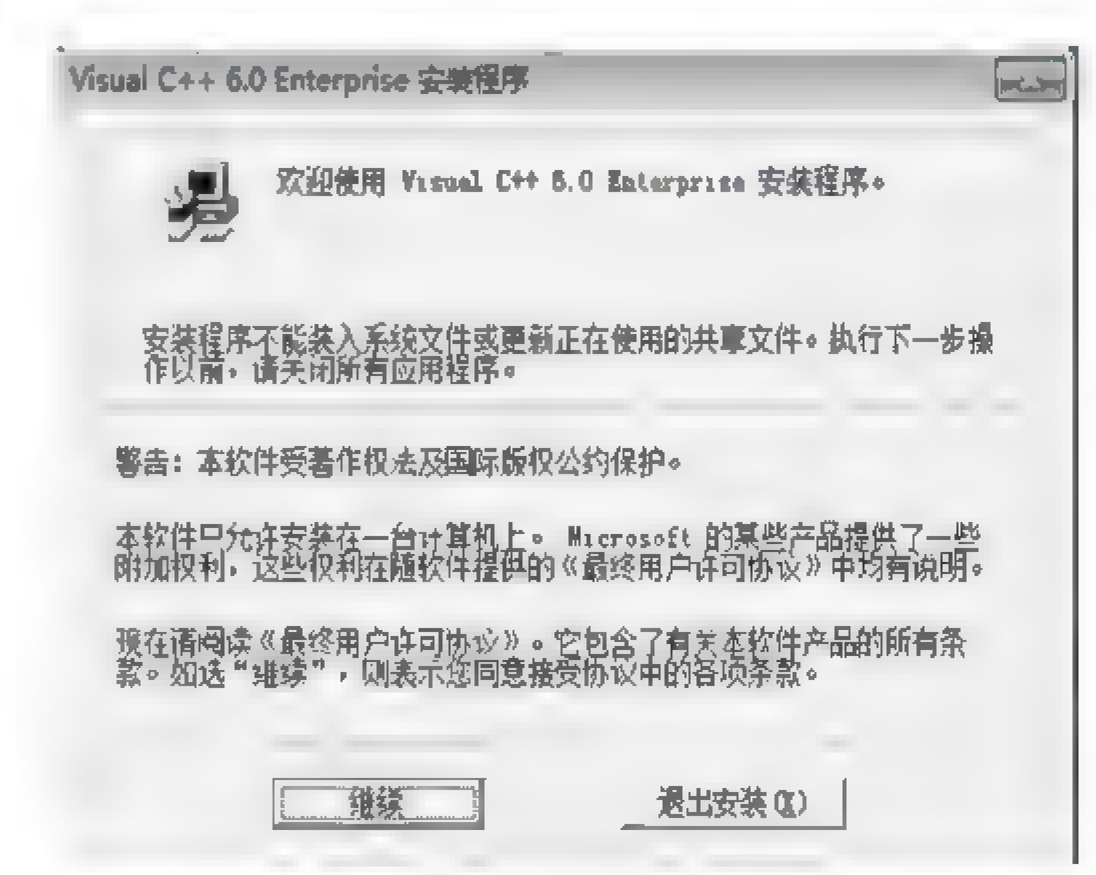


图 1.8　安装程序的欢迎界面

（7）进入产品 ID 确认界面，如图 1.9 所示，在此界面中，显示要安装的 Visual C++6.0 软件的产品 ID，在向 Microsoft 请求技术支持时，需要提供此产品 ID，单击“确定”按钮。

（8）如果读者电脑中安装过 Visual C++6.0，尽管已经卸载了，但是在重新安装时还会提示如图 1.10 所示的信息。安装软件检测到系统之前安装过 Visual C++6.0，如果想要覆盖安装的话，单击“是”按钮；如果要将 Visual C++6.0 安装在其他位置的话，单击“否”按钮。这里单击“是”按钮，继续安装。

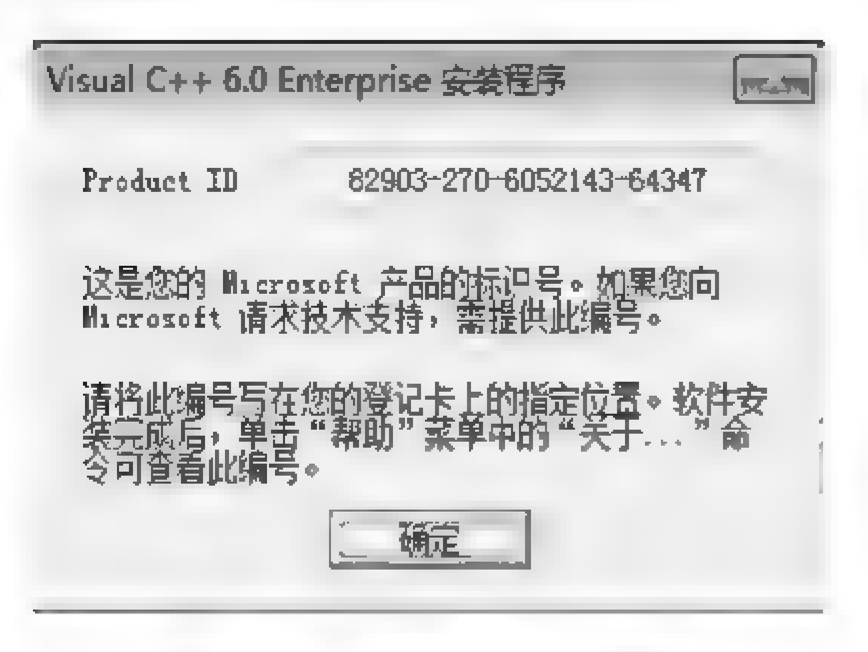


图 1.9　产品 ID 确认界面

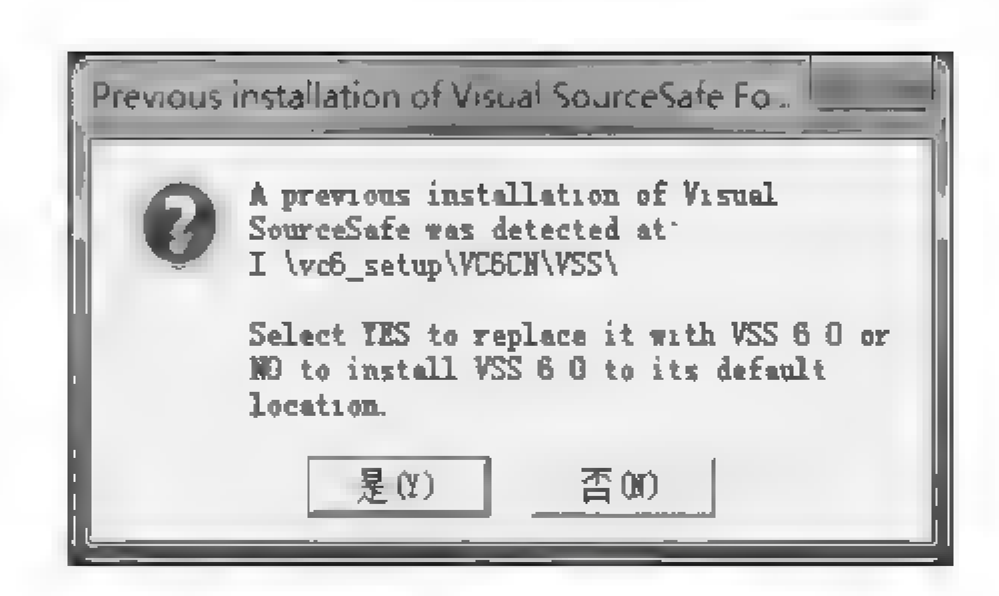


图 1.10　覆盖以前的安装

（9）进入选择安装类型界面，如图 1.11 所示。在此界面中，Typical 为传统安装，Custom 为自定义安装，这里选择 Typical 安装类型。

（10）进入注册环境变量界面，如图 1.12 所示，在此界面中，选中 Register Environment Variables 复选框，注册环境变量，单击 OK 按钮。



图 1.11　选择安装类型界面

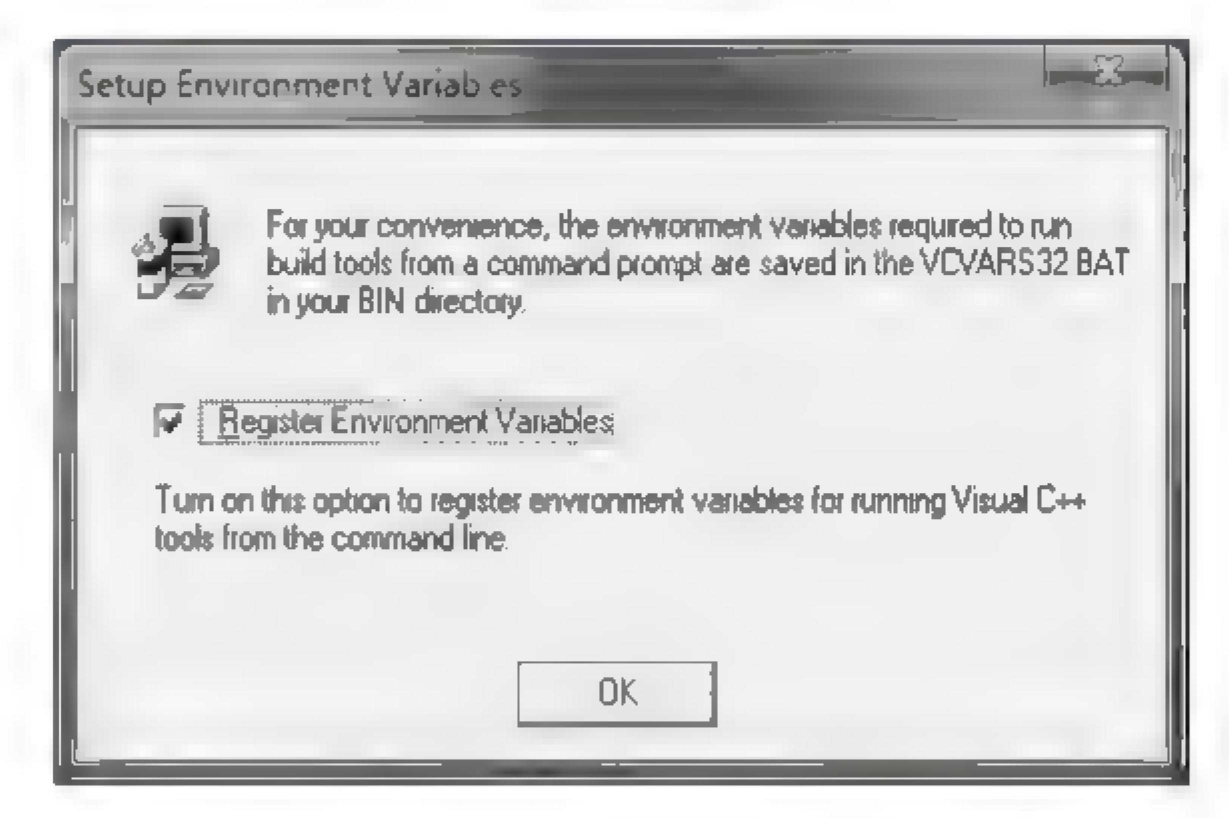


图 1.12　注册环境变量界面

（11）前面的安装选项都设置好之后，下面就开始安装 Visual C++6.0 了，如图 1.13 所示，显示安装进度，当进度条达到 100%时，则安装成功，如图 1.14 所示。

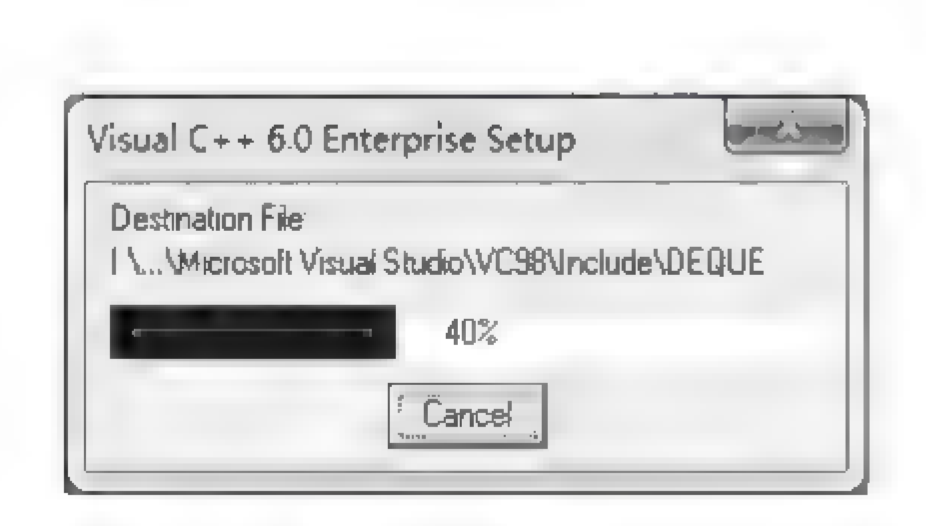


图 1.13　安装进度条



图 1.14　安装成功界面

说明

如果是 Win10 系统，当进度条达到 100%时，将会弹出未响应的界面，这是 Visual C++ 6.0 与 Win10 的兼容性问题，此时只需要双击该界面，在弹出的对话框中单击“关闭程序”按钮即可，然后在电脑的“开始”菜单中找到 Visual C++ 6.0，打开就可以使用。

（12）Visual C++6.0 安装成功后，进入 MSDN 安装界面。取消选中“安装 MSDN”，不安装 MSDN，单击“下一步”按钮。在其他客户工具和服务器安装界面不进行选择，直接单击“下一步”按钮，则可完成 Visual C++6.0 的全部安装。

2．Visual C++ 6.0 的使用

下面通过一个简单的实例来讲解如何使用 Visual C++ 6.0 这个强大的开发工具。

（1）安装 Visual C++ 6.0 之后，选择“开始”菜单中的 Microsoft Visual C++ 6.0 命令，操作如图 1.15 所示。

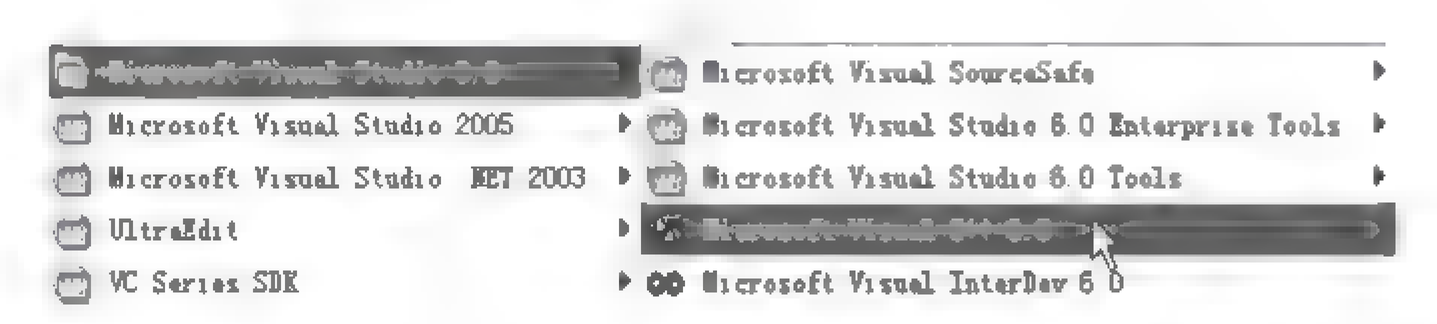


图 1.15　打开 Visual C++ 6.0 开发环境的命令

（2）打开 Visual C++ 6.0 开发环境，进入 Visual C++ 6.0 的界面，如图 1.16 所示。

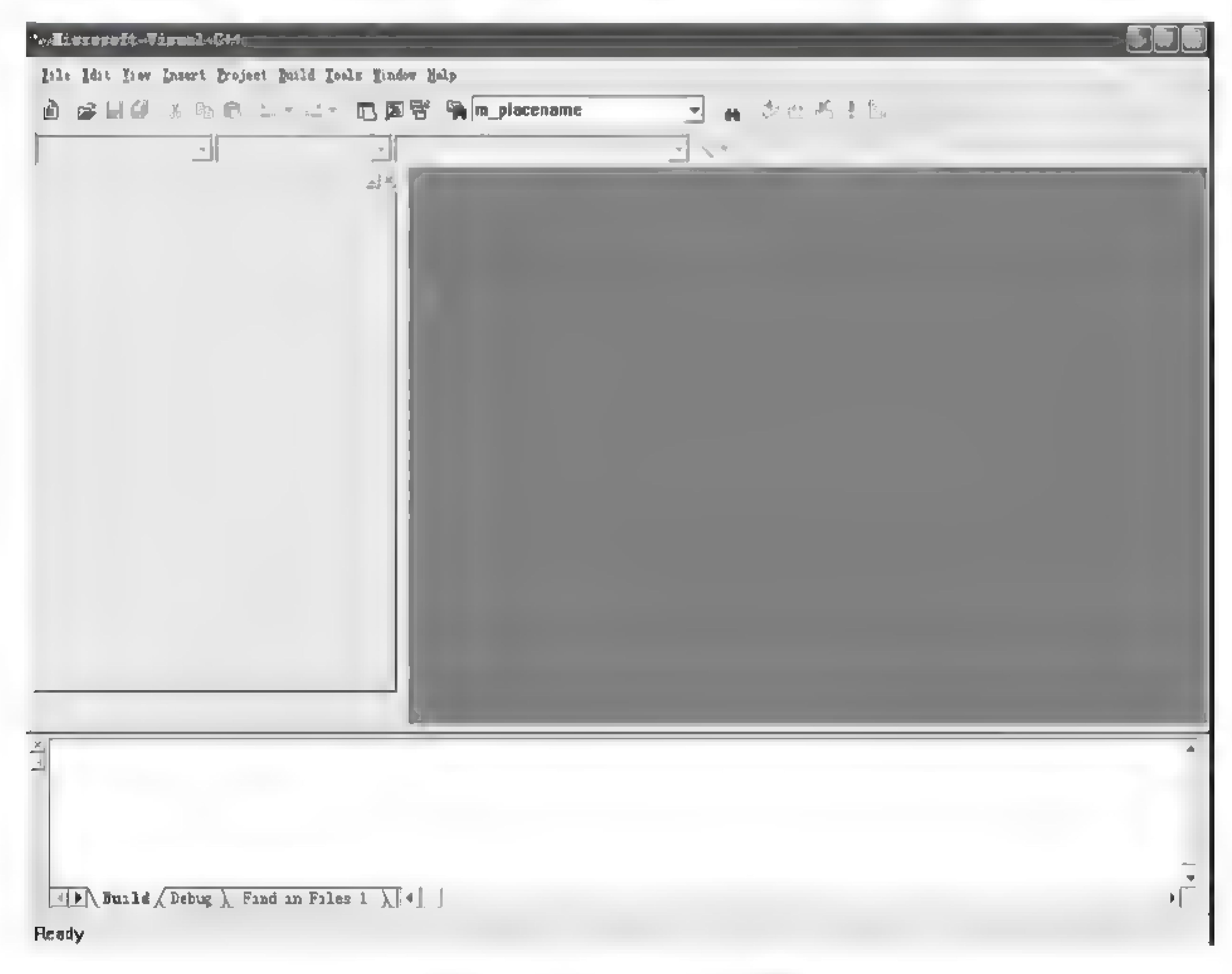


图 1.16　Visual C++ 6.0 界面

（3）在编写程序前，首先要创建一个新的文件，具体方法为：在 Visual C++ 6.0 界面选择 File 菜单中的 New 命令，或者按 Ctrl+N 快捷键，这样就可以创建一个新的文件，如图 1.17 所示。

（4）此时会出现一个选择创建文件的对话框，在此可以选择要创建的文件类型。

要创建一个 C 源文件，首先应选择 Files 选项卡，这时会在列表框中显示可以创建的不同文件。选择其中的 C++ Source File 选项，在右边的 File 文本框中输入要创建的文件名称。



图 1.17　创建一个新文件

注意

因为要创建的是 C 源文件，所以在文本框中要将 C 源文件的扩展名一起输入。例如，创建名称为 Hello 的 C 源文件，那么应该在文本框中输入“Hello.c”。

File 文本框的下面还有一个 Location 文本框，该文本框中是源文件的保存地址，可以通过单击右边的按钮，修改源文件的存储位置。

选择创建文件操作的示意图如图 1.18 所示。

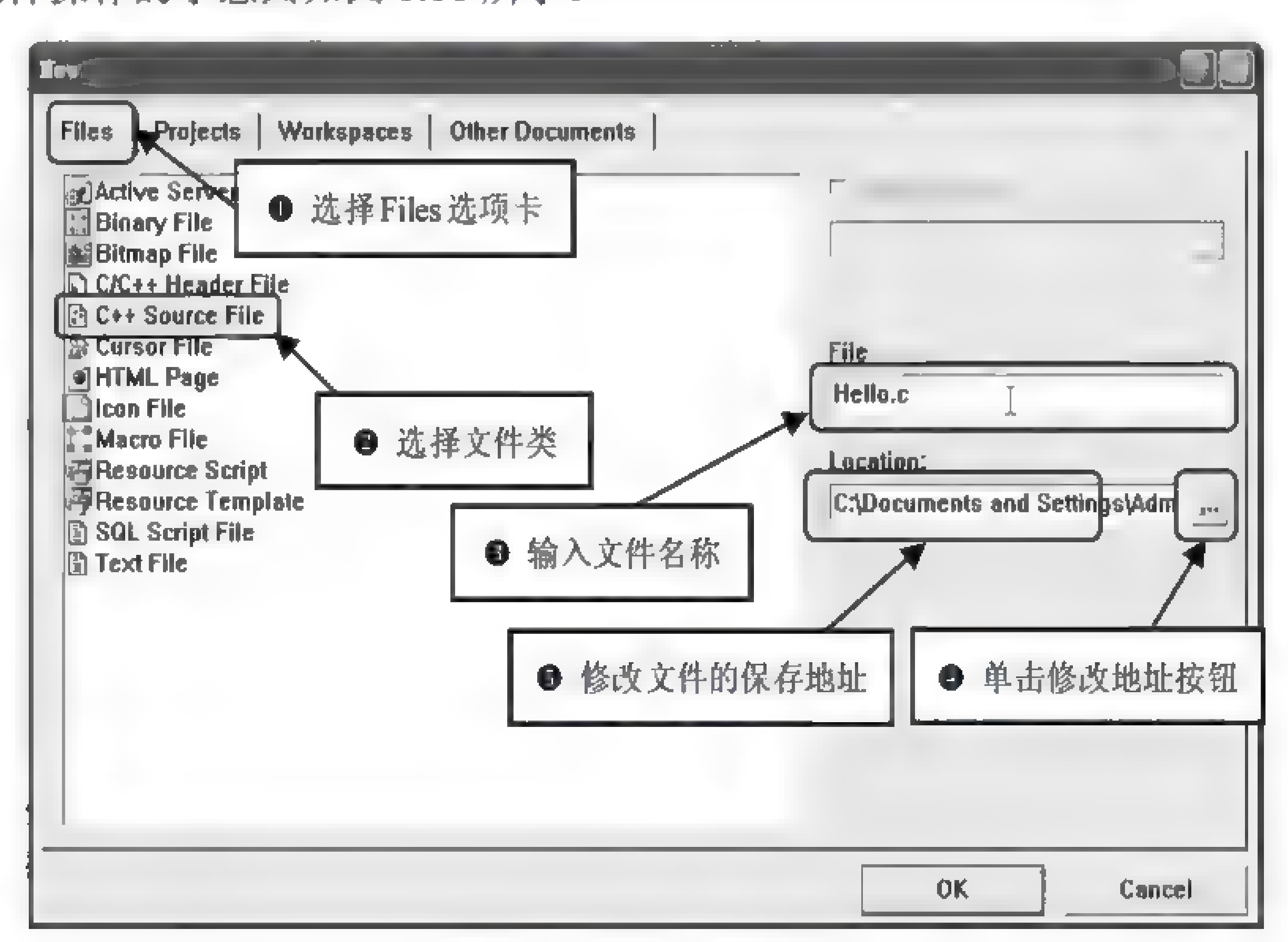


图 1.18　创建 C 源文件

（5）当指定好源文件的保存地址和文件的名称后，单击 OK 按钮，创建一个新的文件。此时可以看到在开发环境中指定创建的 C 源文件，如图 1.19 所示。

（6）C 源文件此时已经创建完成了，现在将一个简单的程序代码输入其中。为了有对比的效果，这里还是使用例 1.1 中的程序。将例 1.1 中的程序输入后的显示效果如图 1.20 所示。

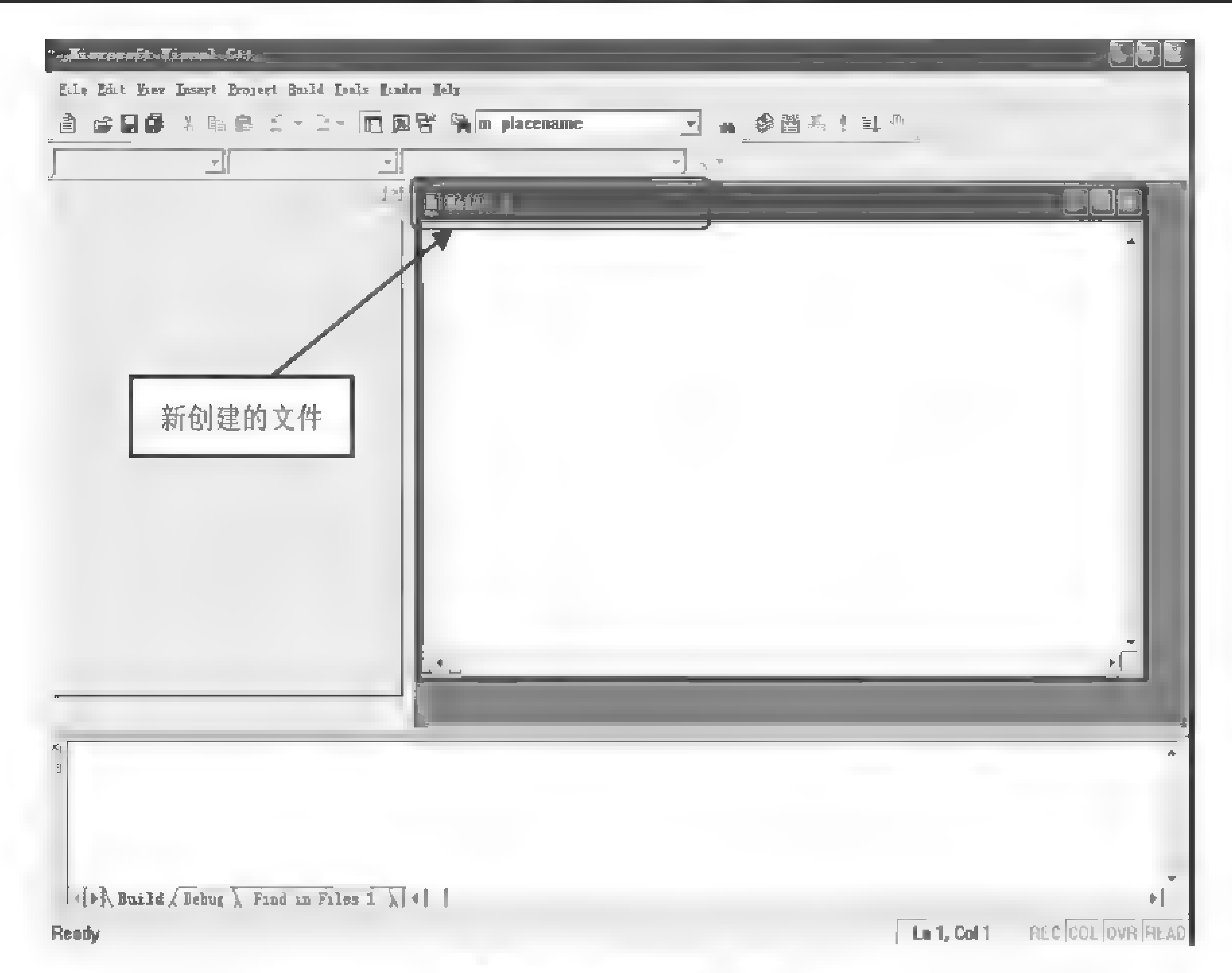


图 1.19　新创建的文件

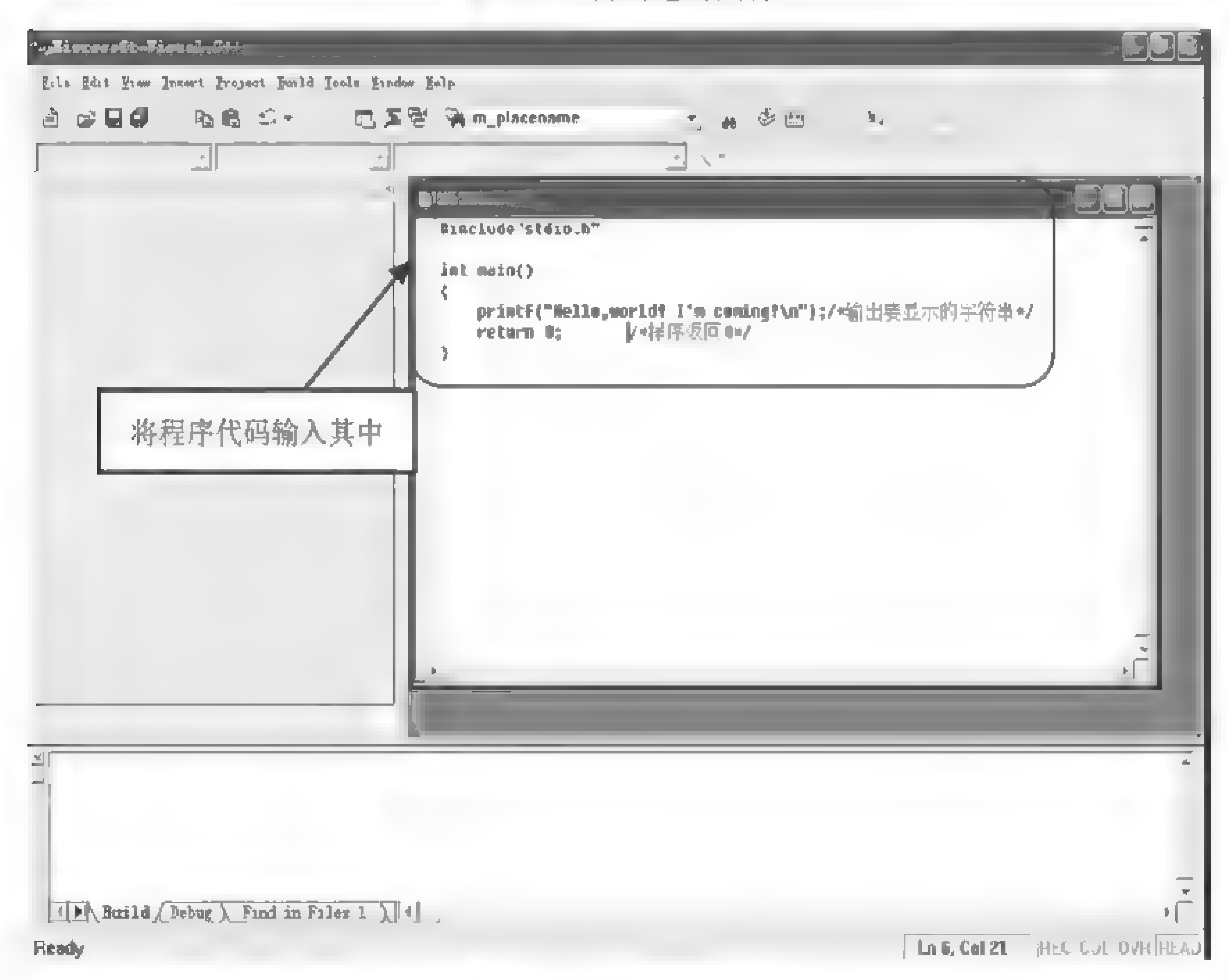


图 1.20　输入程序代码

（7）此时程序已经编写完成，可以对写好的程序进行编译。选择 Build 菜单中的 Compile 命令，如图 1.21 所示。

（8）出现如图 1.22 所示的对话框，询问是否创建一个默认项目工作环境。

（9）单击“是”按钮，此时会询问是否要改动源文件的保存地址，如图 1.23 所示。

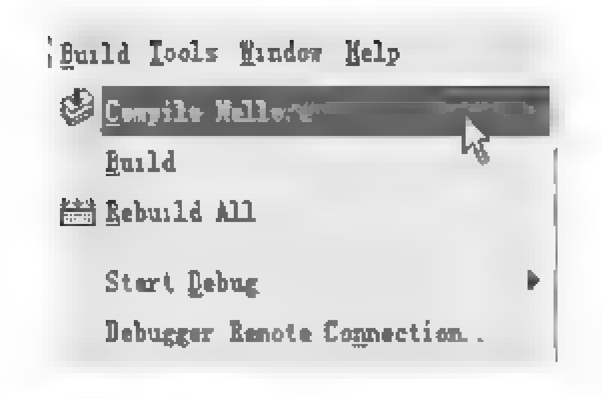


图 1.21　选择 Compile 命令



图 1.22　询问是否创建工作环境

（10）单击“是”按钮后，编译程序。如果程序没有错误，即可被成功编译，虽然此时代码已经被编译，但是还没有链接生成.mp4 可执行文件，因此如果此时要执行程序，会出现如图 1.24 所示的提示对话框，询问是否要创建.mp4 可执行文件。单击“是”按钮，则会链接生成.mp4 文件，即可执行程序。



图 1.23　询问是否要改动源文件的保存地址



图 1.24　询问是否要创建.mp4 文件

（11）当然也有直接创建.mp4 文件的操作选项。可以选择 Build 菜单中的 Build 命令，执行创建.mp4 文件操作，如图 1.25 所示。

注意

在编译程序时可以直接选择 Build 命令进行编译、链接，这样就不用进行上面第(8)步的 Compile 操作，而可以直接将编译和链接操作一起执行。

（12）只有执行程序才可以看到有关程序执行的结果显示，可以选择 Build 菜单中的 Execute 命令进行执行程序操作，即可观察到程序的运行结果，如图 1.26 所示。

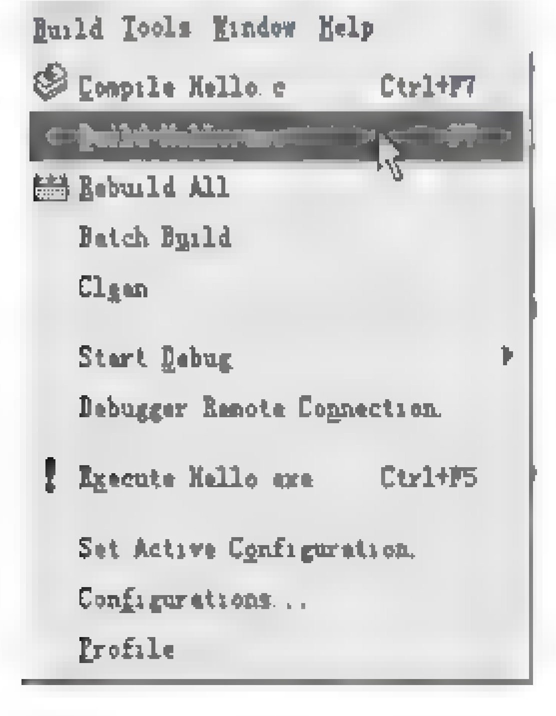


图 1.25　选择 Build 命令

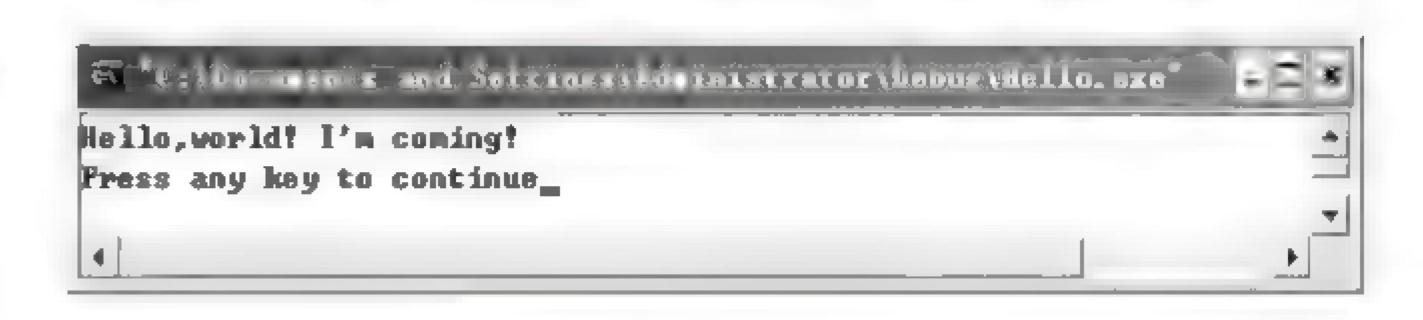


图 1.26　程序运行结果显示

上面通过一个小程序的创建、编辑、编译和显示程序运行结果等操作，介绍了有关使用 Visual C++ 6.0 的简单操作。

下面对 Visual C++ 6.0 集成开发环境的使用进行补充说明。

（1）工具栏按钮的使用

Visual C++ 6.0 集成开发环境提供了如下有用的工具栏按钮。

☑ ：代表 Compile 操作。

☑ ：代表 Build 操作。

☑ ：代表 Execute 操作。

上述工具按钮的功能及作用已经在前面的具体讲解中有所介绍，此处不再赘述。

（2）常用的快捷键

在编写程序时，使用快捷键会加快程序的编写进度。在此建议读者对于常用的操作最好能熟记其快捷键。

☑ Ctrl+N：创建一个新文件。

☑ Ctrl+]：检测程序中的括号是否匹配。

☑ F7：Build 操作。

☑ Ctrl+F5：Execute（执行）操作。

☑ Alt+F8：整理多段不整齐的源代码。

☑ F5：进行调试。

为了便于读者阅读代码，可将程序运行结果的显示底色和文字进行修改。修改过程如下：

（1）按 Ctrl+F5 快捷键执行一个程序，在程序的标题栏上单击鼠标右键，在弹出的快捷菜单中选择“属性”命令，如图 1.27 所示。

（2）此时弹出“属性”对话框，在“颜色”选项卡中对“屏幕文字”和“屏幕背景”进行修改，如图 1.28 所示。在此读者可以根据自己的喜好设定颜色并显示。



图 1.27　选择“属性”命令

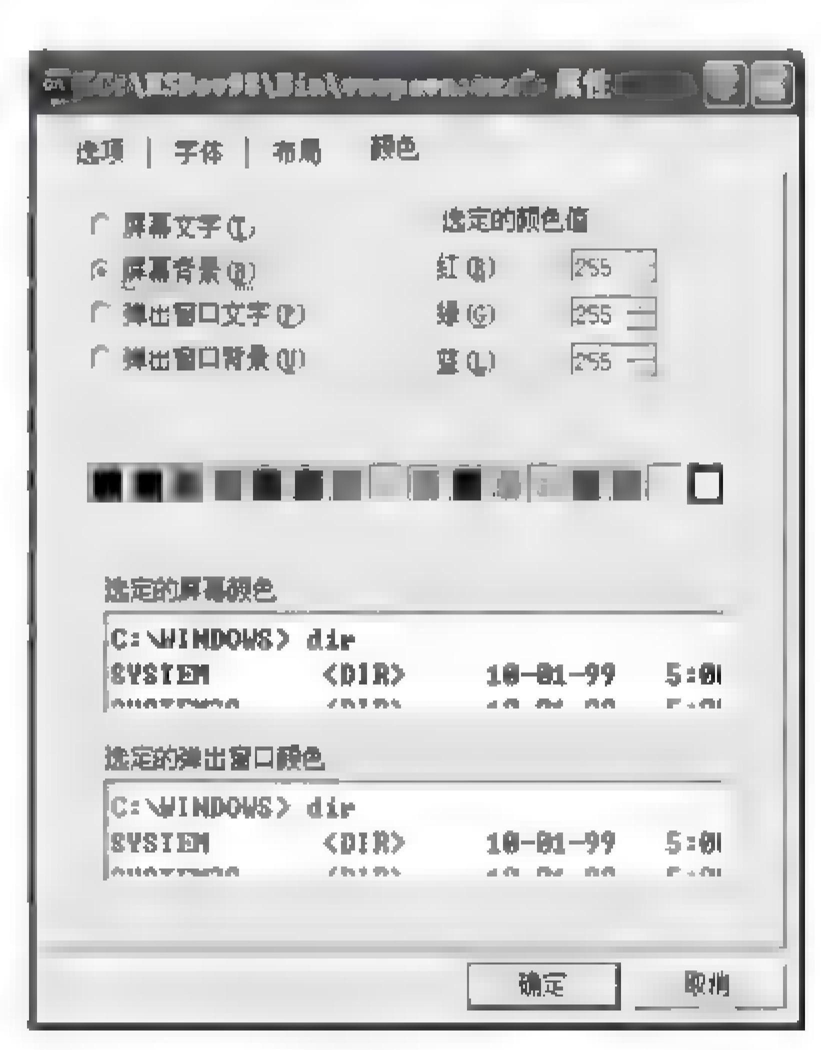


图 1.28　“颜色”选项卡

1.6.2　Visual Studio 2017

Microsoft Visual Studio（简称 VS）是美国微软公司的开发工具包系列产品。Visual Studio 是一个基本完整的开发工具集，用 Visual Studio 编写的代码适用于微软支持的所有平台，Visual Studio 不仅可以编写 C 语言代码，还可以开发 C++、C#、ASP.NET 等，所以 Visual Studio 很强大。Visual Studio 是目前最流行的 Windows 平台应用程序的集成开发环境之一。接下来将介绍 Visual Studio 2017 的安装和使用过程。

1. Visual Studio 2017 的安装

本节以 Visual Studio 2017 社区版的安装为例讲解具体的安装步骤。

说明

Visual Studio 2017 社区版是完全免费的，其下载地址为 https://www.visualstudio.com/zh-hans/downloads/。

安装 Visual Studio 2017 社区版的步骤如下：

（1）Visual Studio 2017 社区版的安装文件是 exe 可执行文件，其命名格式为“vs_community__编译版本号.mp4”，笔者在写作本书时，下载的安装文件名为 vs_community__1230733315.1531385802.mp4 文件，双击该文件开始安装。

说明

安装 Visual Studio 2017 开发环境时，计算机上要求必须安装了.NET Framework 4.6 框架，如果没有安装，请先到微软官方网站下载并安装，下载地址为 https://www.microsoft.com/zh-CN/download/details.aspx?id=48130。

（2）程序首先跳转到 Visual Studio 2017 安装程序界面，在该界面中单击“继续”按钮，随即自动跳转到安装选择项界面，如图 1.29 所示，在该界面中主要将“使用 C++的桌面开发”复选框选中，其他复选框，读者可以根据自己的开发需要确定是否选择安装；选择完要安装的功能后，在下面“位置”处选择要安装的路径，这里建议不要安装在系统盘上，可以选择一个其他磁盘进行安装，例如，这里笔者将其安装到了 D 盘。设置完成后，单击“安装”按钮。

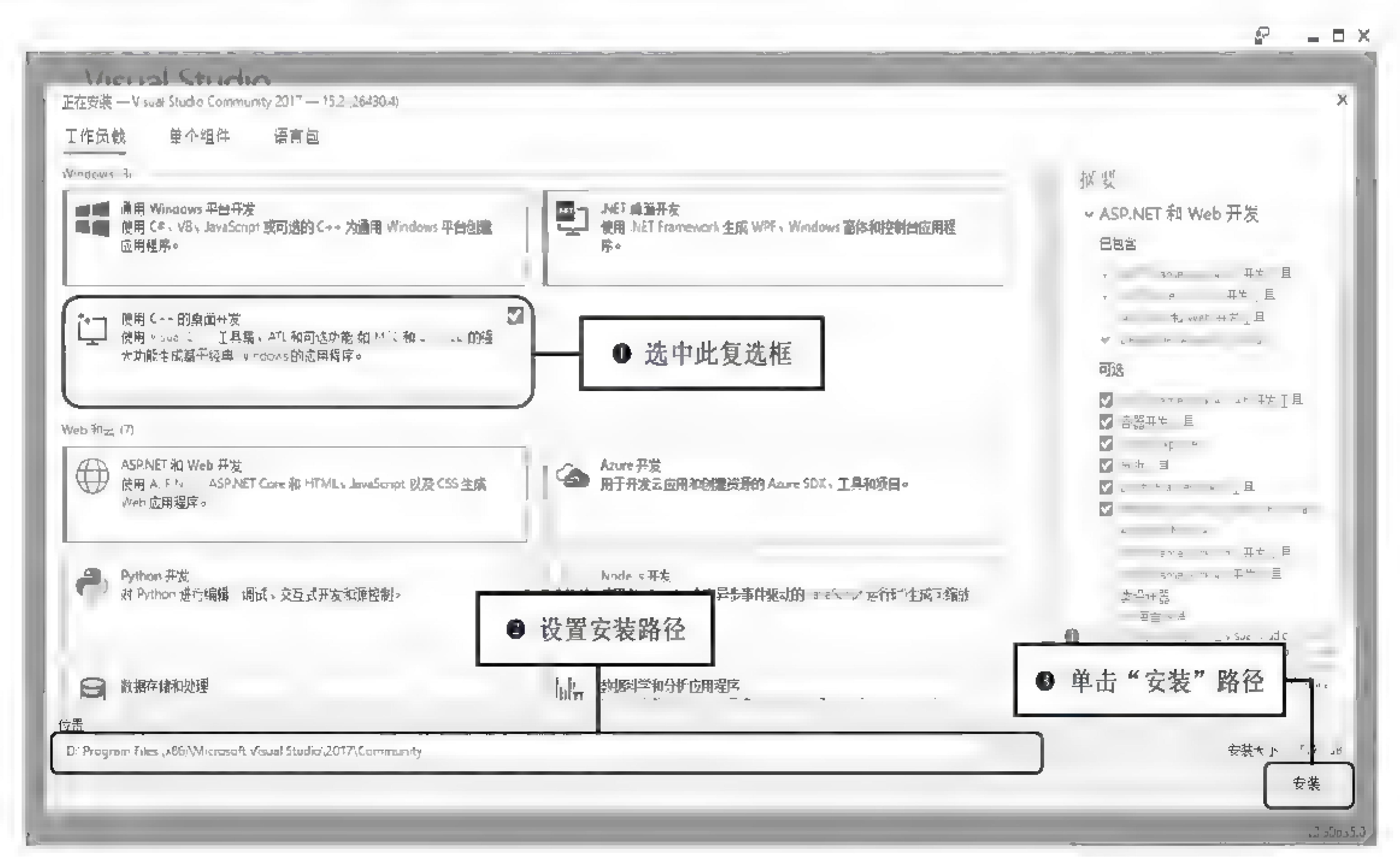


图 1.29　Visual Studio 2017 安装界面



注意

在安装 Visual Studio 2017 开发环境时，一定要确保计算机处于联网状态，否则无法正常安装。

（3）跳转到如图 1.30 所示的安装进度界面，该界面显示当前的安装进度，等待安装进度条完成后，自动进入安装完成页，此时就可以在系统的开始菜单中选择 Visual Studio 2017 菜单来启动并使用开发环境了。



图 1.30　Visual Studio 2017 安装界面

（4）安装完成后，也就是进度条为 100%时，就会出现如图 1.31 所示的界面。单击“重启”按钮，完成 Visual Studio 2017 的安装。



图 1.31　重启界面

（5）重启计算机后，在 Windows 的“开始”菜单中找到 Visual Studio 2017 的开发环境，选择 Visual Studio 2017，如果是第一次打开 Visual Studio 2017，会出现欢迎界面，直接单击“以后再说”按钮。

（6）进入 Visual Studio 2017 环境的开发设置界面，如图 1.32 所示，在“开发设置”下拉列表框中选择 Visual C++，颜色根据自己的喜好来选择，笔者选了蓝色，最后单击“启动 Visual Studio”按钮。

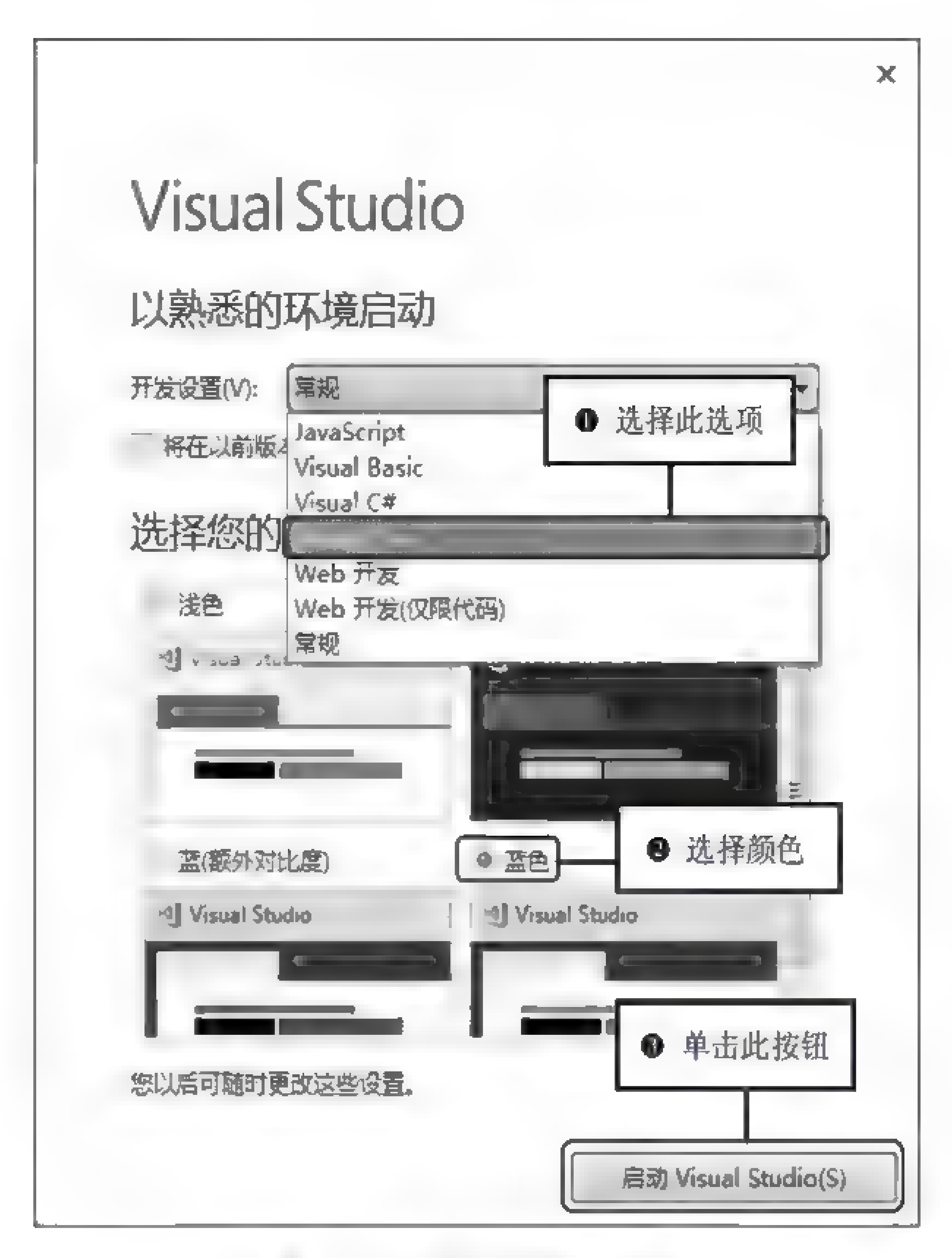


图 1.32 选择开发设置界面

（7）进入 Visual Studio 2017 环境启动界面。等待几秒钟后，进入 Visual Studio 2017 环境开发的欢迎界面。

2. Visual Studio 2017 的使用

安装好了 Visual Studio 2017 开发环境，接下来使用 Visual Studio 2017 创建一个项目，具体步骤如下：

（1）打开 Visual Studio 2017 环境后出现欢迎界面，在编写程序之前，首先需要创建一个新程序文件，具体方法是：在 Visual Studio 2017 欢迎界面中选择“文件”→“新建”→“项目”命令，如图 1.33 所示，或者按 Shift+Ctrl+N 组合键进入“新建项目”对话框。



图 1.33 创建一个新文件

（2）在“新建项目”对话框中选择要创建的文件夹类型。选择创建文件操作的过程如图 1.34 所示。

首先选择“Windows 桌面”选项，这时在右侧列表框中将显示可以创建的不同类型的文件夹，这里选择[redacted]选项，在“名称”文本框中输入要创建的文件夹名称，如 Dome。在“位置”下拉列表框中设置文件夹的保存地址，可以通过单击右边的 浏览(B)... 按钮修改源文件的存储位置。

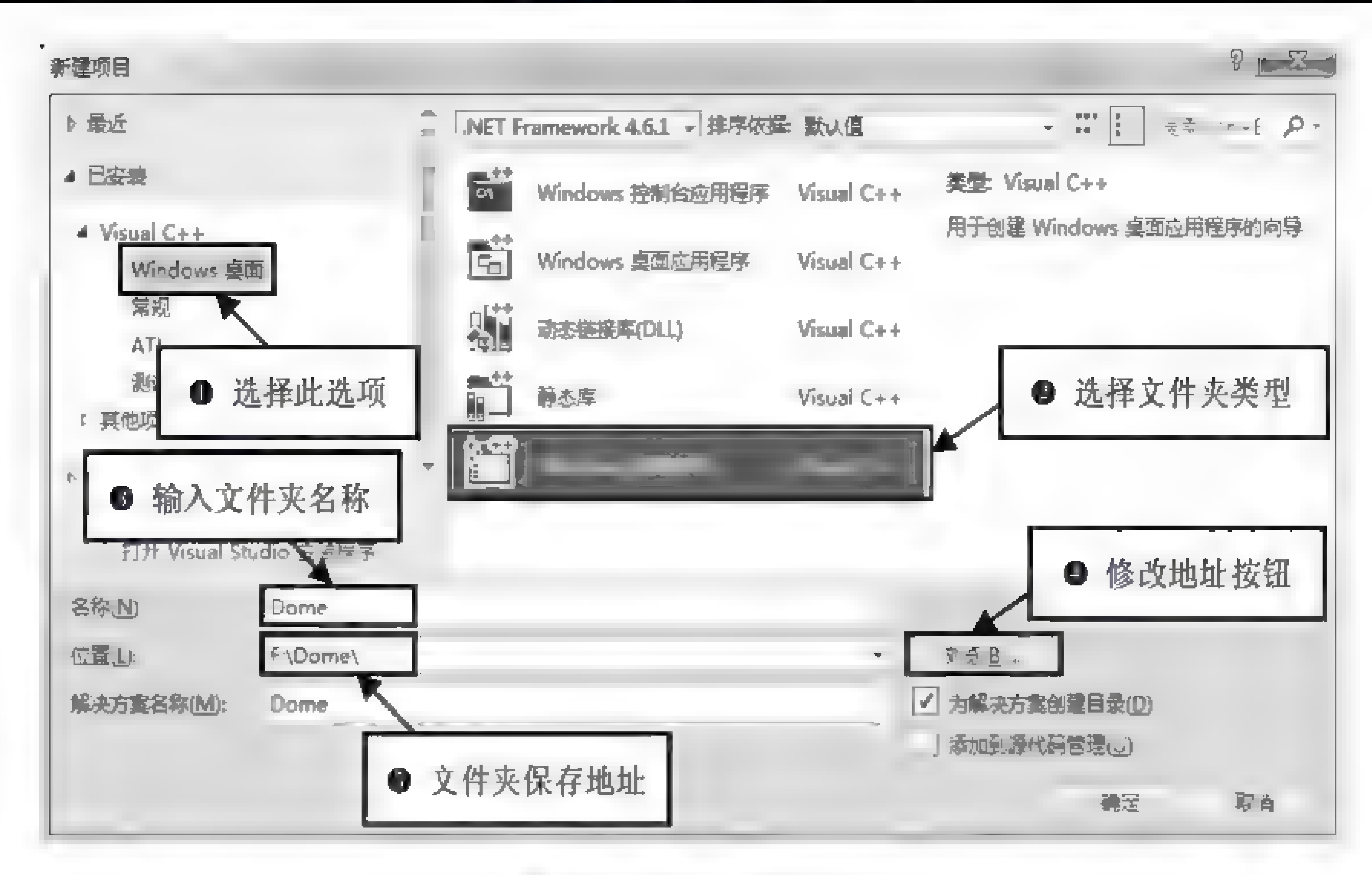


图 1.34　创建 C 源文件

（3）指定好文件夹的保存地址和名称后，单击“确定”按钮，会弹出如图 1.35 所示界面，选中“空项目”复选框，然后单击“确定”按钮，自动跳转到如图 1.36 所示的界面。

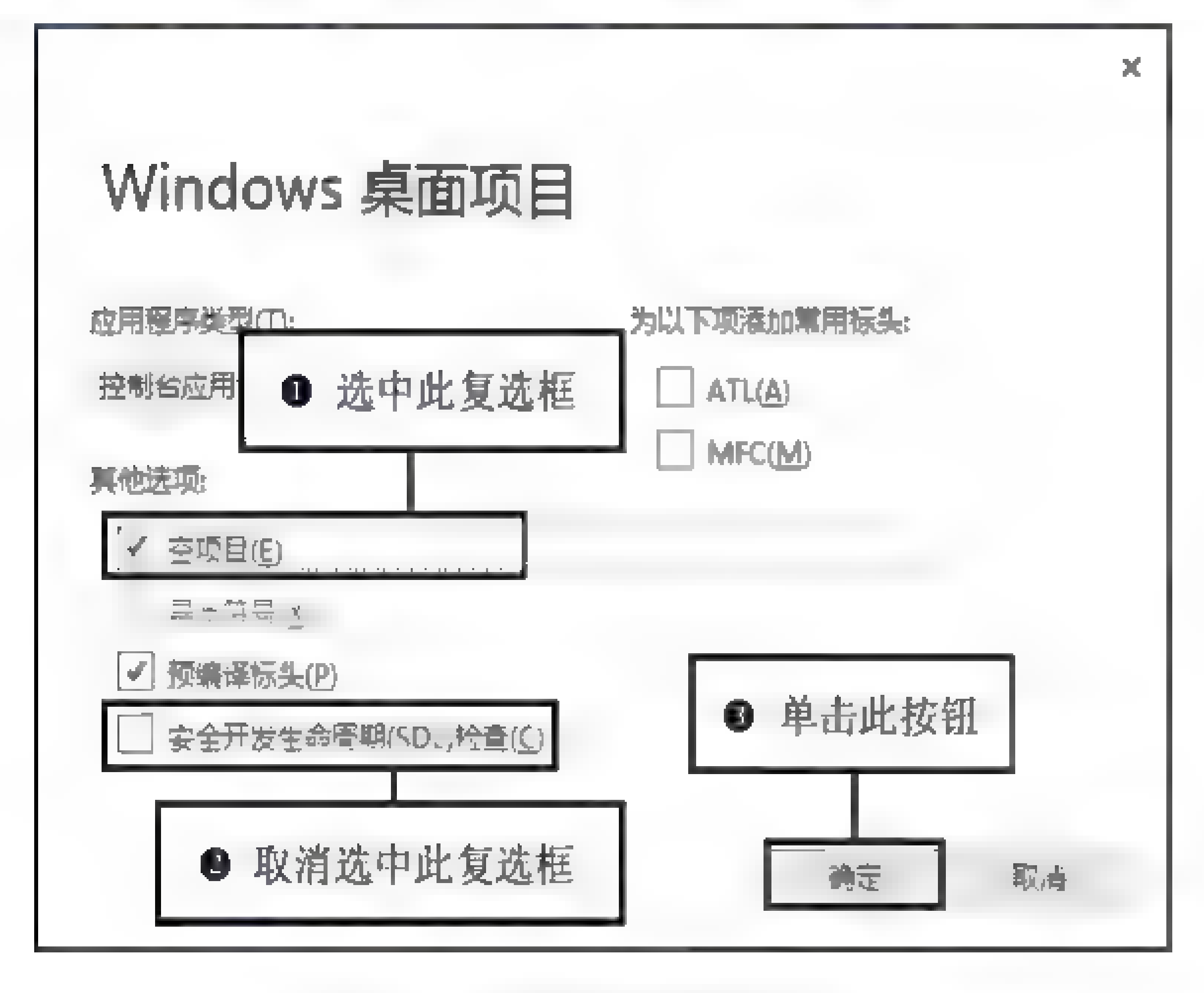


图 1.35　创建应用程序向导



图 1.36　创建项目界面

（4）选择“解决方案资源管理器”中的源文件，右击“源文件”，在弹出的快捷菜单中选择“添加”→“新建项”命令，如图 1.37 所示，或者按 Shift+Ctrl+A 组合键进入添加项目界面。



图 1.37　添加项目界面

（5）完成步骤（4）就会自动跳转到如图 1.38 所示的窗口。

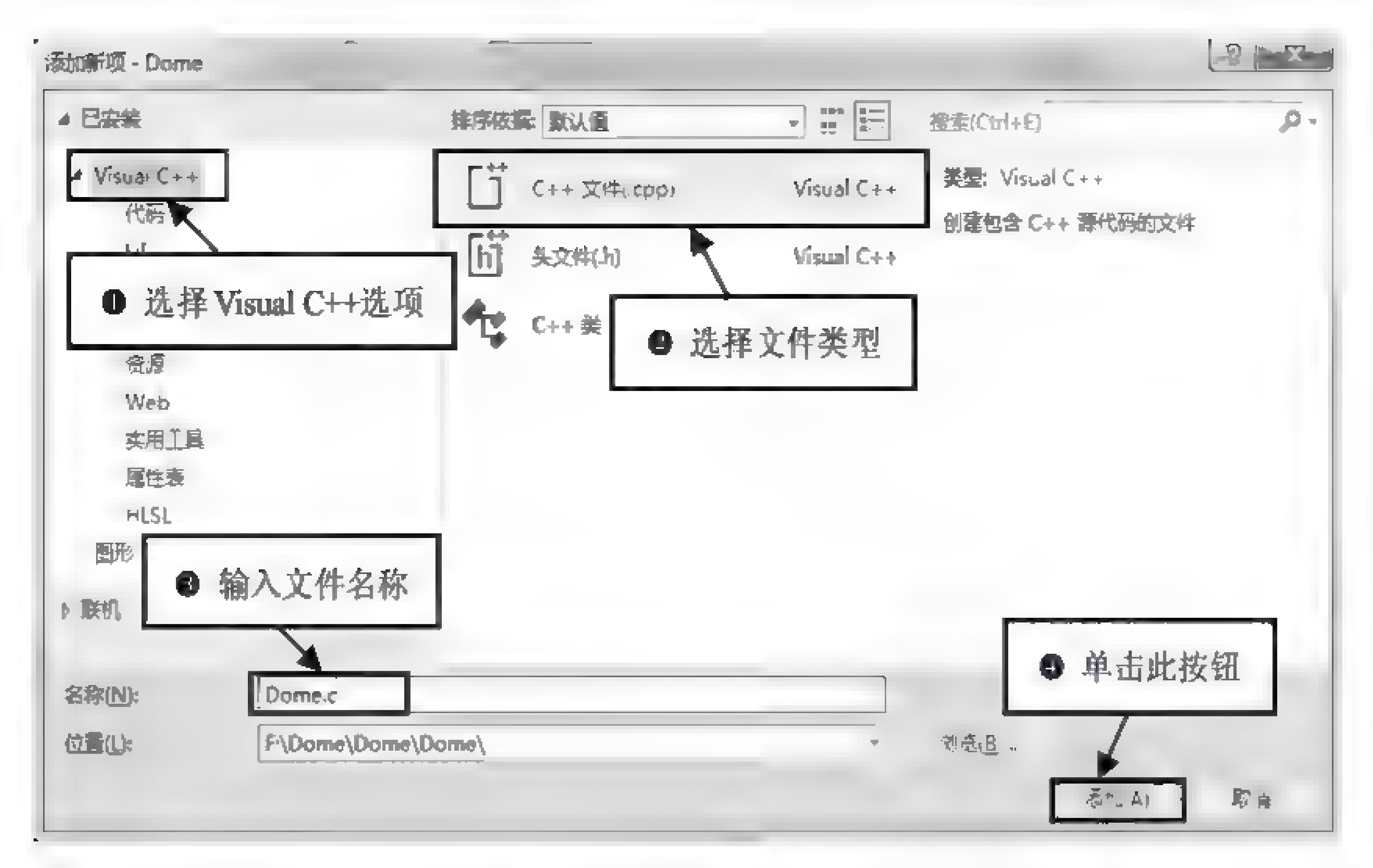


图 1.38　添加项目界面

添加项目时首先选择 Visual C++选项，这时在右侧列表框中将显示可以创建的不同文件。因为要创建 C 文件，因此这里选择[redacted]选项，在下方的“名称”文本框中输入要创建的 C 文件名称，如 Dome.c。“位置”下拉列表框是文件夹的保存地址，这里默认为步骤（2）创建的文件夹位置，不做更改。

注意

因为要创建的是 C 源文件，所以在文本框中要将默认的扩展名.cpp 改为.c。例如创建名称为 Dome 的 C 源文件，那么应该在文本框中显示“Dome.c”。

（6）单击“添加”按钮，这样就添加了一个 C 文件，如图 1.39 所示。



图 1.39　完成添加 C 文件

将代码写入 Dome.c 文件中，如图 1.40 所示。

（7）代码编写完之后，接下来就要编译程序了。在 Visual Studio 2017 菜单栏中选择“生成”→“编译”命令，或者按 Ctrl+F7 快捷键编译程序，如图 1.41 所示。

如果编译程序之后，在输出工作空间的位置输出“生成：成功 1 个，失败 0 个，最新 0 个，跳过 0 个”表示编译成功。

（8）程序已经编译成功，并且成功地生成了可执行文件，接下来就是运行程序了。在 Visual Studio 2017 的菜单栏中选择“调试”→“开始执行（不调试）”命令，或者按 Ctrl+F5 快捷键，如图 1.42 所示。

运行出如图 1.43 所示的结果。

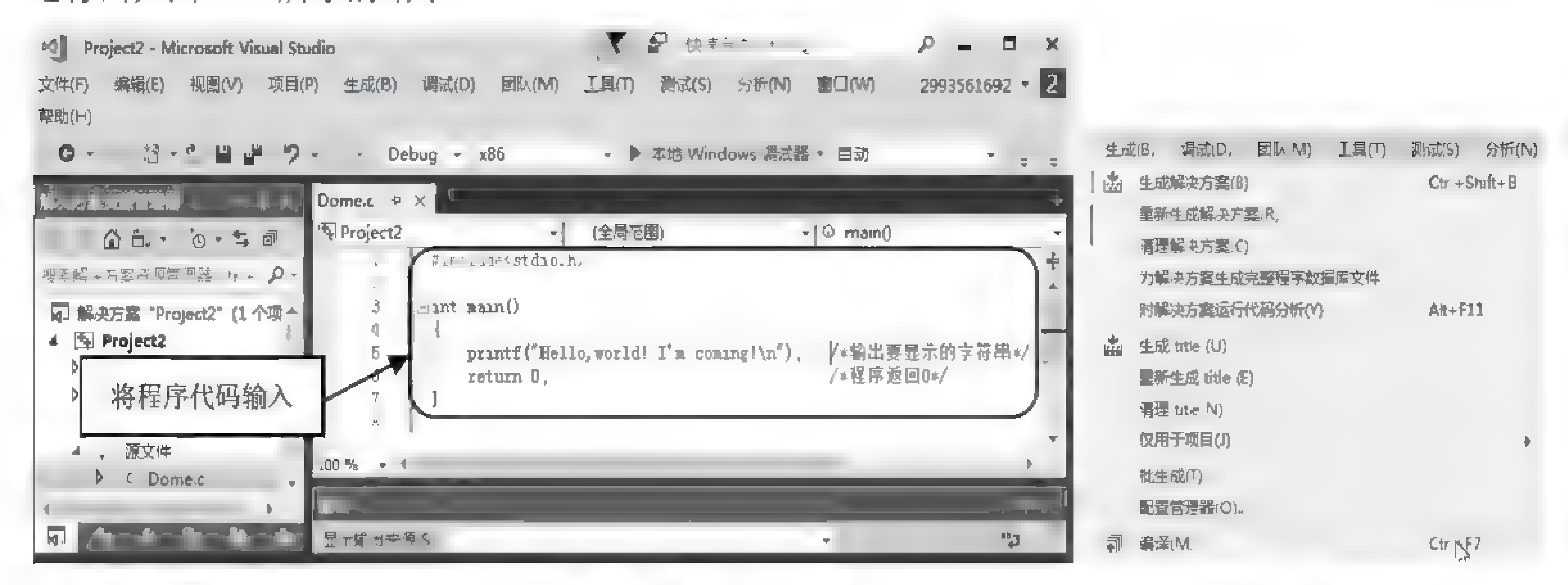


图 1.40　输入代码　　　　图 1.41　编译程序

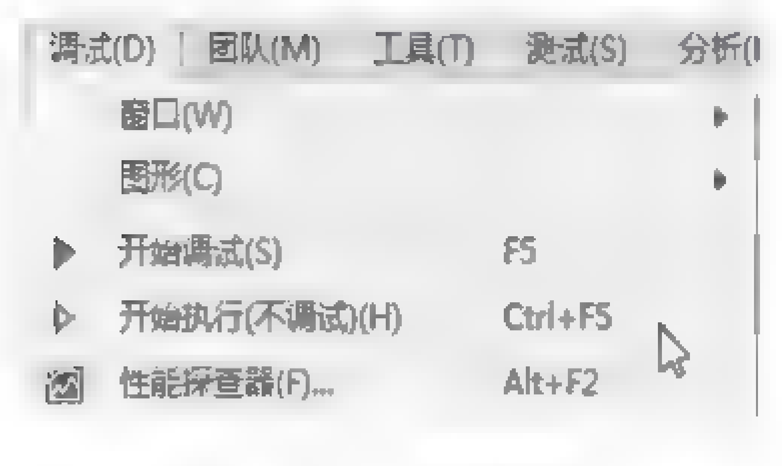


图 1.42　运行程序

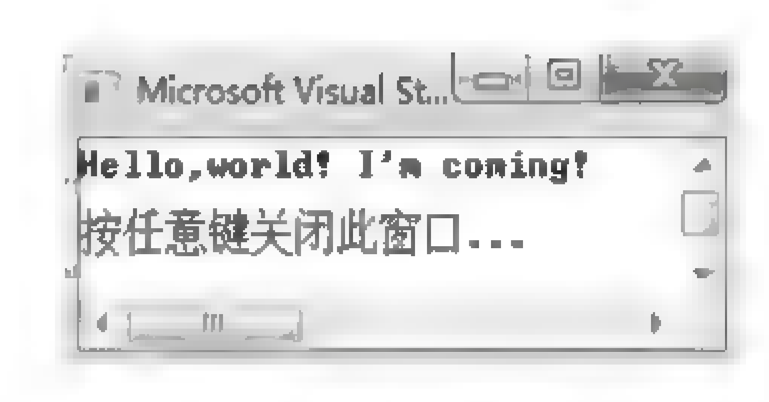
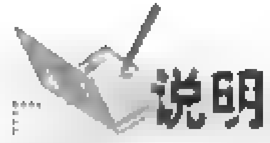


图 1.43　运行结果



说明

如果您觉得程序没有错误，可以直接运行程序。

1.7　小　　结

本章首先讲解了关于 C 语言的发展历史，可以看出 C 语言的重要性。然后讲解了 C 语言的特点，通过这些特点进一步验证了 C 语言的重要地位。接下来通过一个简单的 C 语言程序和一个完整的 C 语言程序，将 C 语言的概貌呈现给读者，使读者对 C 语言编程有一个总体的认识。最后对两个比较流行的 C 程序开发环境进行了介绍，通过实例的创建，对如何使用这两种集成开发环境进行了详细的说明，使读者按书中的步骤就可以编写实现自己的程序，为后面的学习提供了验证程序结果的方法，并且培养了动手实践的能力。

第 2 章

算法

（视频讲解：22 分钟）

一个程序通常包含算法、数据结构、程序设计方法以及语言工具和环境这 4 个方面。其中，算法是核心，解决的是“做什么”和“如何做”的问题。正是因为算法如此重要，所以这里单独列出一章来介绍算法的基本知识。

通过阅读本章，您可以：

- 了解算法的特性
- 了解如何用自然语言描述算法
- 掌握如何用 3 种基本结构表示算法
- 掌握 N-S 流程图



2.1　算法的基本概念

算法与程序设计以及数据结构密切相关，是解决一个问题的完整的步骤描述，是解决问题的策略、规则和方法。算法的描述形式有很多种，如传统流程图、结构化流程图及计算机程序语言等，下面就介绍算法的一些相关内容。

2.1.1　算法的特性

算法是为解决某一特定类型的问题而制定的一个实现过程，它具有下列特性。

1. 有穷性

一个算法必须在执行有穷步之后结束，且每一步都可在有穷时间内完成，不能无限地执行下去。如要编写一个由小到大整数累加的程序，这时要注意一定要设一个整数的最上限，也就是加到哪个数为止。若没有最上限，那么程序将无终止地运行下去，也就是常说的死循环。

2. 确定性

算法的每一个步骤都应当是确切定义的，对于每一个过程不能有二义性，必须对将要执行的每个动作做出严格而清楚的规定。

3. 可行性

算法中的每一步都应当能有效地运行，也就是说算法是可执行的，并要求最终得到正确的结果。如下面一段程序：

```
int x,y,z;
scanf("%d,%d,%d",&x,&y,&z);
if(y==0)
z=x/y;
```

在这段代码中，“z=x/y;”就是一个无效的语句，因为 0 不可以做分母。

4. 输入

一个算法应有零个或多个输入，输入是在执行算法时需要从外界取得的一些必要的（如算法所需的初始量等）信息。例如：

```
int a,b,c;
scanf("%d,%d,%d",&a,&b,&c);
```

上面的代码就是有多个输入。又如：

```
main()
{
```

```
printf("hello world!");
}
```

上面的代码需要零个输入。

5. 输出

一个算法有一个或多个输出。什么是输出？输出就是算法最终所求的结果。编写程序的目的就是要得到一个结果，如果一个程序运行下来没有任何结果，那么这个程序本身也就失去了意义。

2.1.2 算法的优劣

衡量一个算法的好坏，通常要从以下几个方面来分析。

1. 正确性

正确性是指所写的算法应能满足具体问题的要求，即对任何合法的输入，算法都会得出正确的结果。

2. 可读性

可读性是指算法被写好之后，该算法被理解的难易程度。一个算法可读性的好坏十分重要，如果一个算法比较抽象，难以理解，那么这个算法就不易于进行交流和推广使用，其后续修改、扩展、维护都十分不方便。因此在写一个算法时，要尽量将该算法写得简明、易懂。

3. 健壮性

一个程序完成后，运行该程序的用户对程序的理解各有不同，并不能保证每一个人都能按照要求进行输入。健壮性就是指当输入的数据非法时，算法也会做出相应判断，而不会因为输入的错误造成瘫痪。

4. 时间复杂度与空间复杂度

简单地说，时间复杂度就是算法运行所需要的时间。不同的算法具有不同的时间复杂度，当一个程序较小时，会感觉不到时间复杂度的重要性；但当一个程序特别大时，时间复杂度实际上是十分重要的。因此，如何写出更高速的算法一直是算法不断改进的目标。空间复杂度是指算法运行所需的存储空间的多少。随着计算机硬件的发展，空间复杂度已经不再显得那么重要。



2.2 算法的描述

算法包含算法设计和算法分析两个方面。算法设计主要研究怎样针对某一特定类型的问题设计出求解步骤，算法分析则要讨论所设计出来的算法步骤的正确性和复杂性。

对于一些问题的求解步骤，需要一种表达方式，即算法描述。他人可以通过这些算法描述来了解算法设计者的思路。表示一个算法，可以用不同的方法，常用的有自然语言、流程图、N-S 流程图等。下面对算法的描述做进一步介绍。

2.2.1 自然语言

自然语言就是人们日常用的语言，这种表示方式通俗易懂，下面通过实例具体介绍。

【例2.1】 求n!。

(1) 定义3个变量i、n及mul，为i和mul均赋初值为1。

(2) 从键盘中输入一个数，赋给n。

(3) 将mul乘以i的结果赋给mul。

(4) i的值加1，判断i的值是否大于n，如果大于n，则执行步骤(5)，否则执行步骤(3)。

(5) 将mul的结果输出。

【例2.2】 任意输入3个数，求这3个数中的最小数。

(1) 定义4个变量，分别为x、y、z以及min。

(2) 输入大小不同的3个数，分别赋给x、y、z。

(3) 判断x是否小于y，如果小于，则将x的值赋给min，否则将y的值赋给min。

(4) 判断min是否小于z，如果小于，则执行步骤(5)，否则将z的值赋给min。

(5) 将min的值输出。

以上介绍的例2.1和例2.2的算法实现过程就是采用自然语言来描述的。从上面的描述中会发现用自然语言描述的好处，就是易懂。但是采用自然语言进行描述也有很大的弊端，即容易产生歧义。例如，将例2.1步骤(3)中的"将mul乘以i的结果赋给mul"改为"mul等于i乘以mul"，这样就产生了歧义。并且，用自然语言来描述较为复杂的算法时，会显得不是很方便，因此一般情况下不采用自然语言来描述。

2.2.2 流程图

流程图是一种传统的算法表示法，它用一些图框来代表各种不同性质的操作，用流程线来指示算法的执行方向。由于它直观形象，易于理解，所以应用广泛。特别是在语言发展的早期阶段，只有通过流程图才能简明地表述算法。

1. 流程图符号

流程图使用一些图框来表示各种操作。如图2.1所示为一些常见的流程图符号，其中，起止框用来标识算法的开始和结束；判断框用于对一个给定的条件进行判断，根据条件成立与否来决定如何执行后续操作；连接点用于将画在不同地方的流程线连接起来。下面通过一个实例来介绍这些图框应如何使用。

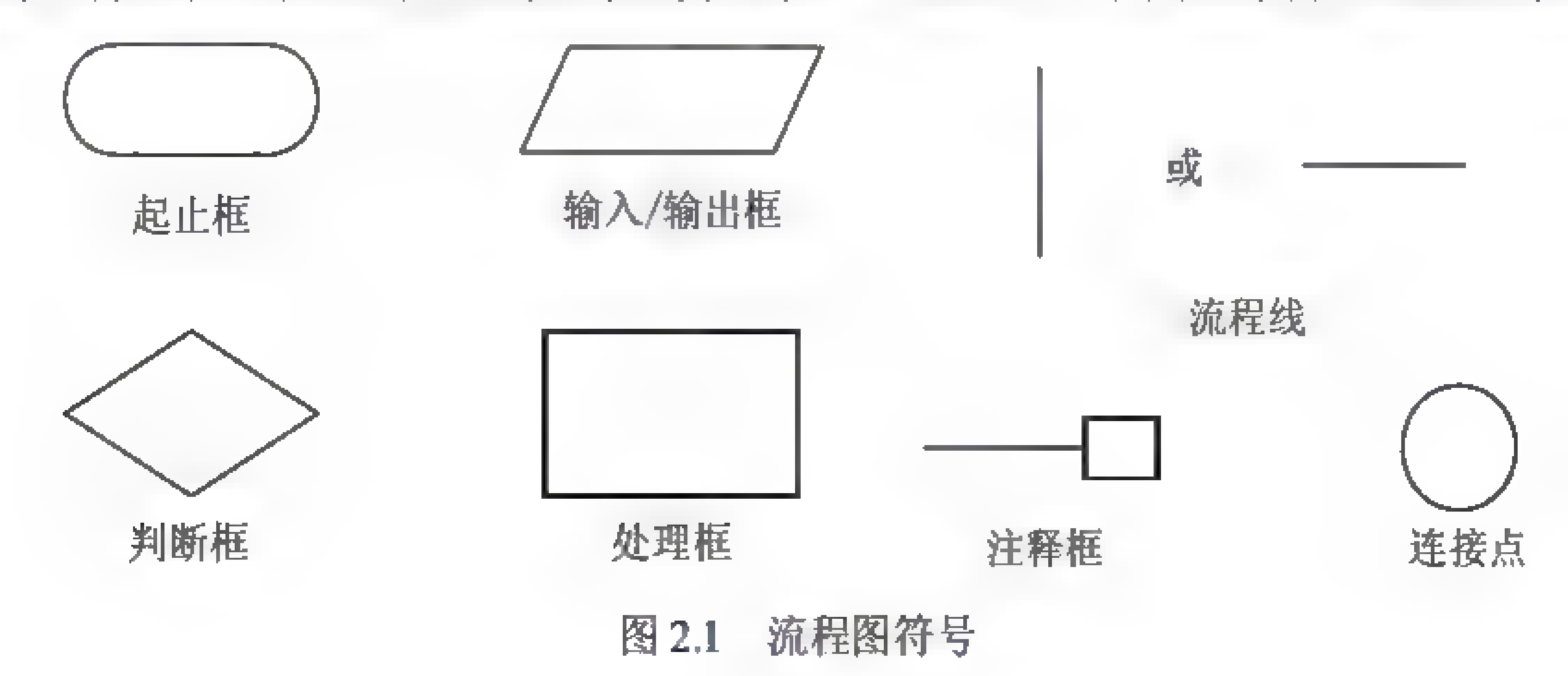


图2.1 流程图符号

2. 3 种基本结构

Bohra 和 Jacopini 为了提高算法的质量，经研究提出了 3 种基本结构，即顺序结构、选择结构和循环结构，因为任何一个算法都可由这 3 种基本结构组成。这 3 种基本结构之间可以并列，可以相互包含，但不允许交叉，不允许从一个结构直接转到另一个结构的内部去。

整个算法都是由 3 种基本结构组成的，所以只要规定好 3 种基本结构的流程图的画法，就可以画出任何算法的流程图。

（1）顺序结构

顺序结构是简单的线性结构，在顺序结构的程序中，各操作是按照它们出现的先后顺序执行的，如图 2.2 所示。

在执行完 A 框所指定的操作后，接着执行 B 框所指定的操作，这个结构中只有一个入口点 A 和一个出口点 B。

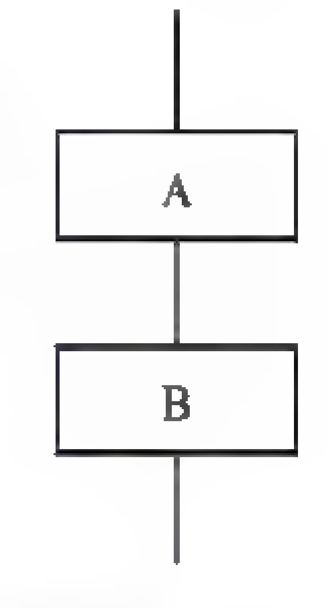


图 2.2　顺序结构

【例 2.3】　输入两个数并分别赋给变量 i 和 j，再将这两个数分别输出。

本实例的流程图可以采用顺序结构来实现，如图 2.3 所示。

（2）选择结构

选择结构也称为分支结构，如图 2.4 所示。

选择结构中必须包含一个判断框。图 2.4 所代表的含义是根据给定的条件 P 是否成立选择执行 A 框还是 B 框。

图 2.5 所代表的含义是根据给定的条件 P 进行判断，如果条件成立则执行 A 框，否则什么也不做。

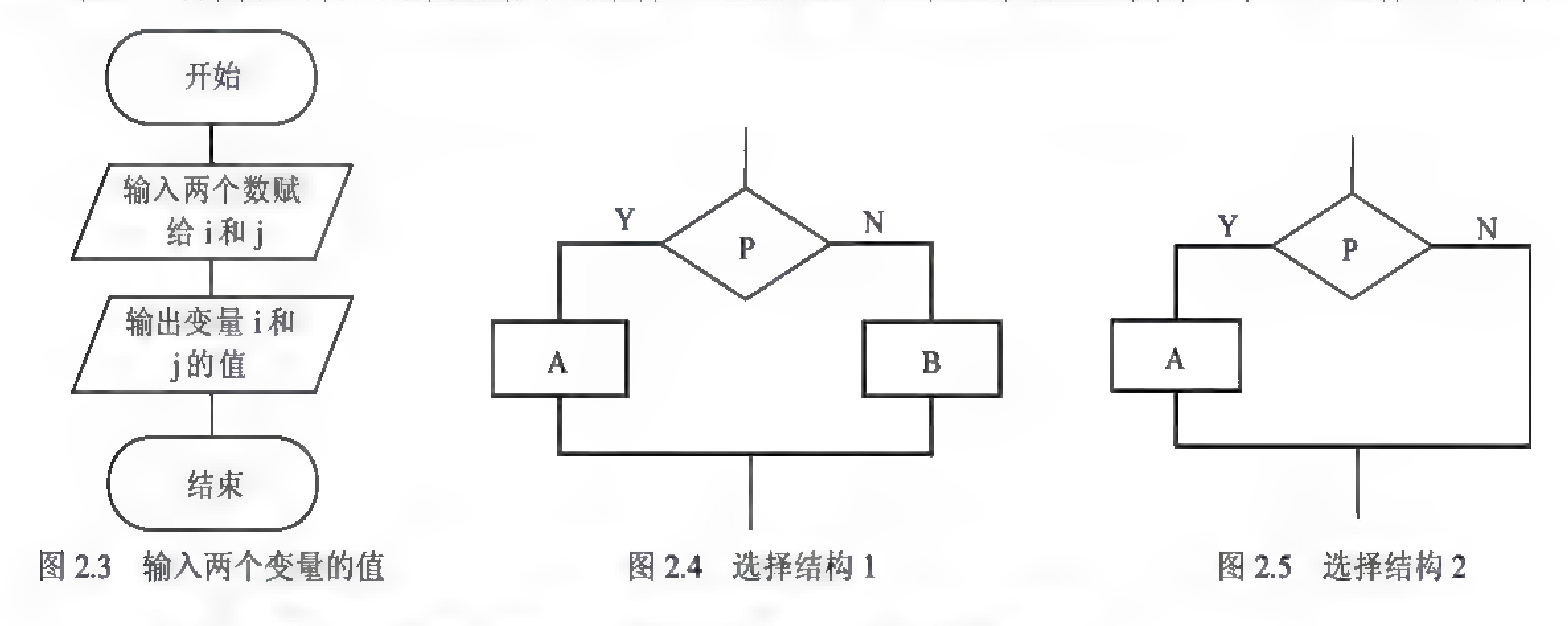


图 2.3　输入两个变量的值　　图 2.4　选择结构 1　　图 2.5　选择结构 2

【例 2.4】　输入一个数，判断该数是否为偶数，并给出相应提示。

本实例的流程图可以采用选择结构来实现，如图 2.6 所示。

（3）循环结构

在循环结构中，反复地执行一系列操作，直到条件不成立时才终止循环。按照判断条件出现的位置，可将循环结构分为当型循环结构和直到型循环结构。

当型循环如图 2.7 所示。当型循环是先判断条件 P 是否成立，如果成立，则执行 A 框；执行完 A 框后，再判断条件 P 是否成立，如果成立，接着再执行 A 框；如此反复，直到条件 P 不成立为止，此时不执行 A 框，跳出循环。

直到型循环如图 2.8 所示。直到型循环是先执行 A 框，然后判断条件 P 是否成立，如果条件 P 成

立则再执行 A；然后判断条件 P 是否成立，如果成立，接着再执行 A 框；如此反复，直到条件 P 不成立，此时不执行 A 框，跳出循环。

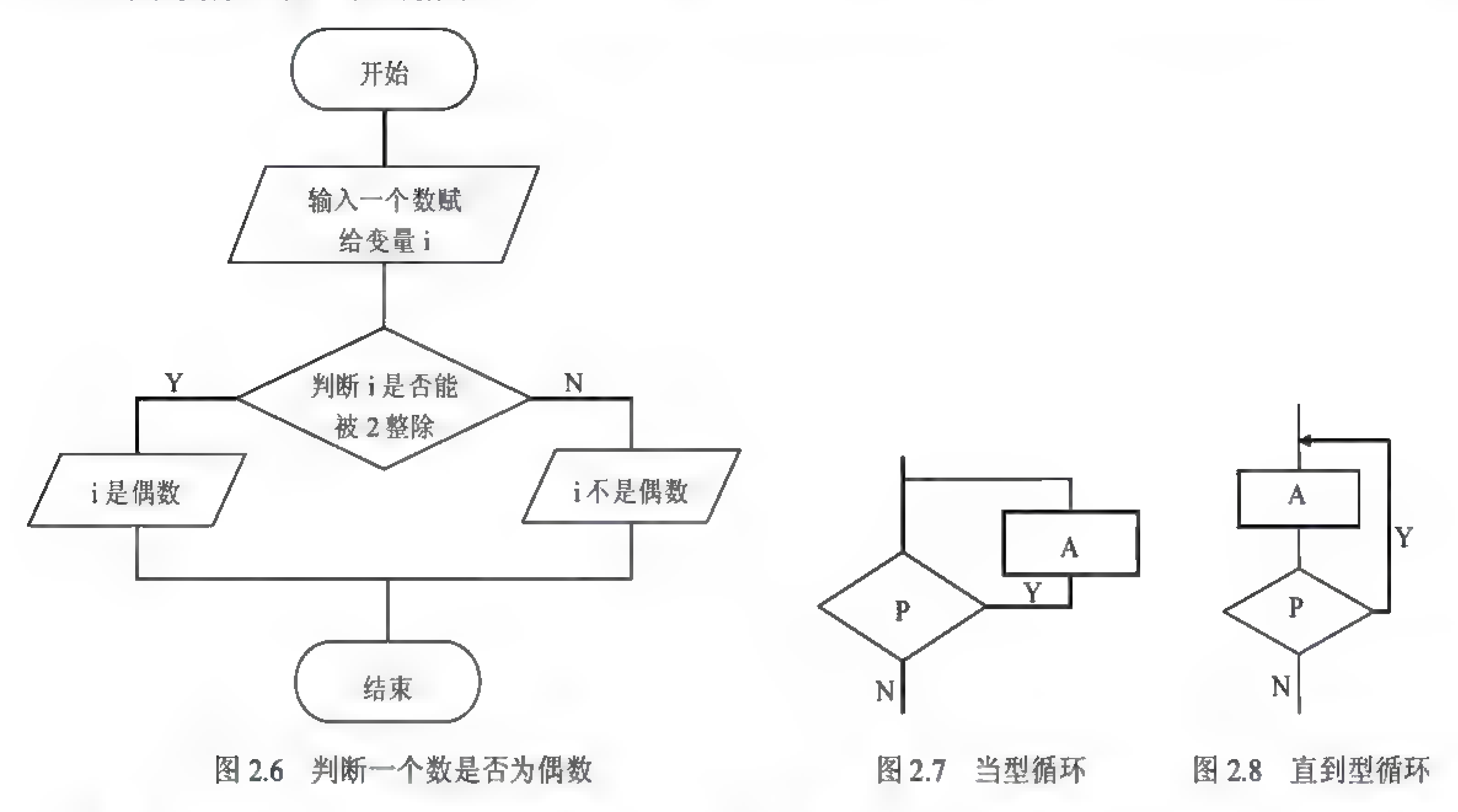


图 2.6　判断一个数是否为偶数　　图 2.7　当型循环　　图 2.8　直到型循环

【例 2.5】　求 1 和 100 之间（包括 1 和 100）所有整数之和。

本实例的流程图可以用当型循环结构来表示，如图 2.9 所示。也可以用直到型循环结构来表示，如图 2.10 所示。

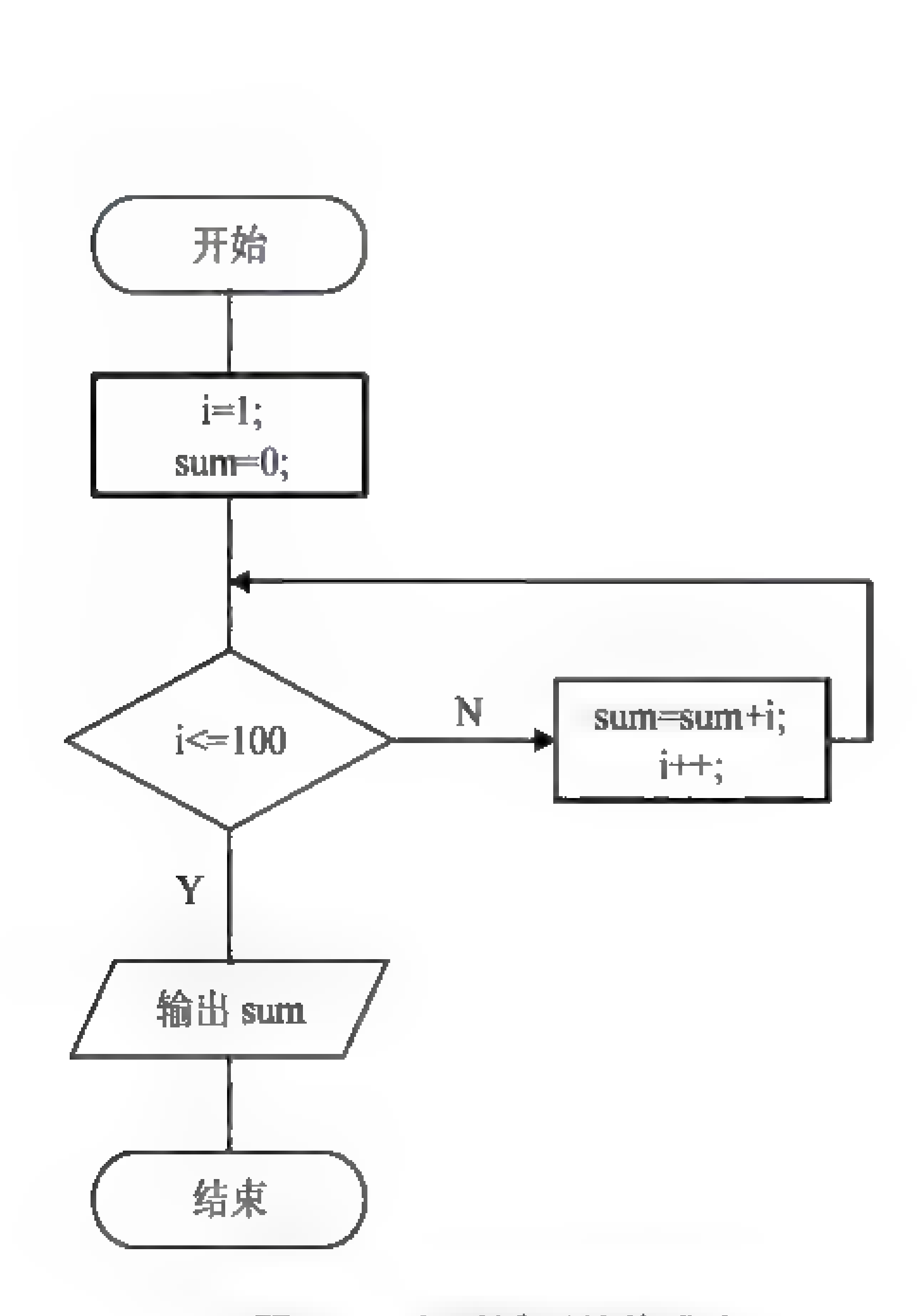


图 2.9　当型循环结构求和

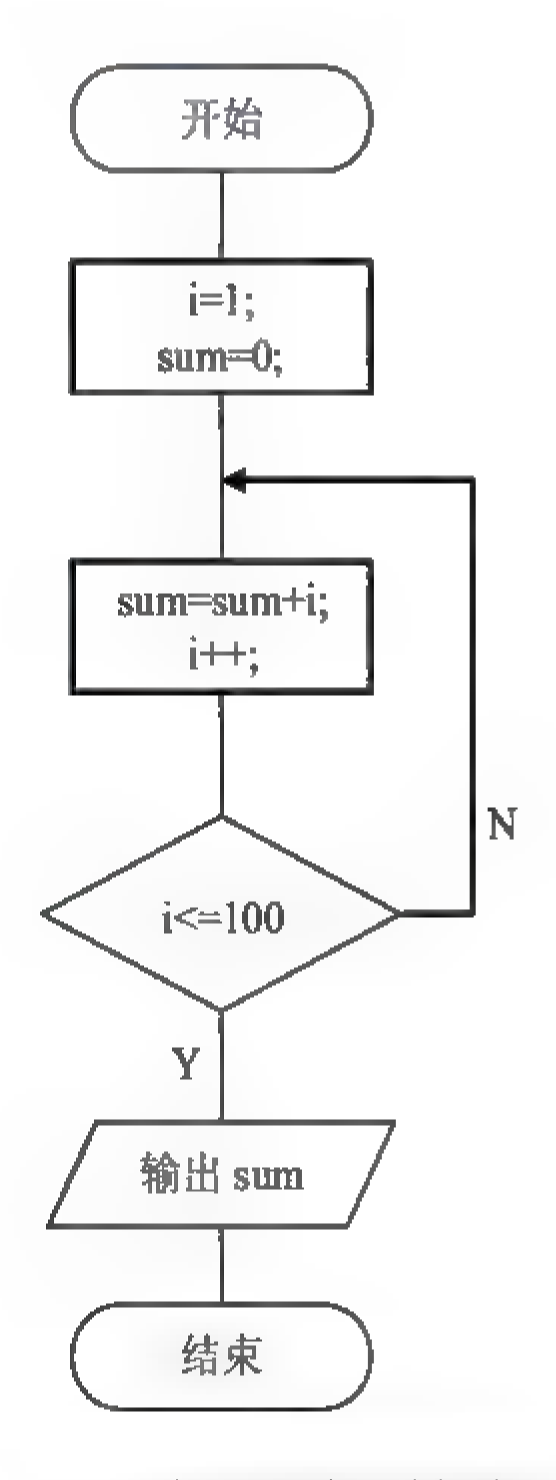


图 2.10　直到型循环结构求和

2.2.3 N-S 流程图

N-S 流程图是另一种算法表示法，是由美国人 I.Nassi 和 B.Shneiderman 共同提出的，其根据是：既然任何算法都是由前面介绍的 3 种结构组成的，则各基本结构之间的流程线就是多余的，因此去掉了所有流程线，将全部的算法写在一个矩形框内。N-S 图也是算法的一种结构化描述方法，同样也有 3 种基本结构，下面分别进行介绍。

1. 顺序结构

顺序结构的 N-S 流程图如图 2.11 所示。例 2.3 的 N-S 流程图如图 2.12 所示。

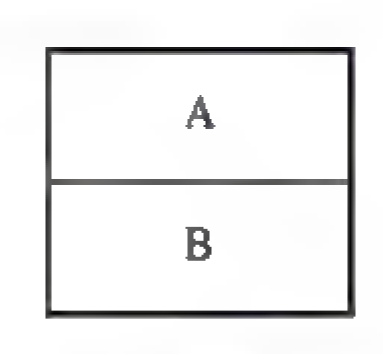


图 2.11　顺序结构

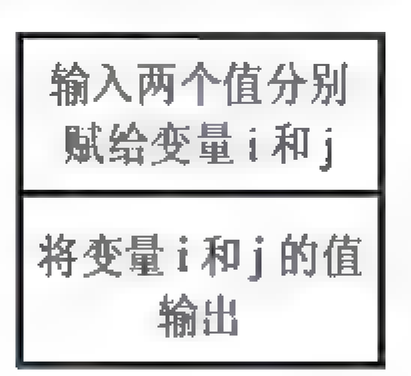


图 2.12　输出变量的值

2. 选择结构

选择结构的 N-S 流程图如图 2.13 所示。例 2.4 的 N-S 流程图如图 2.14 所示。

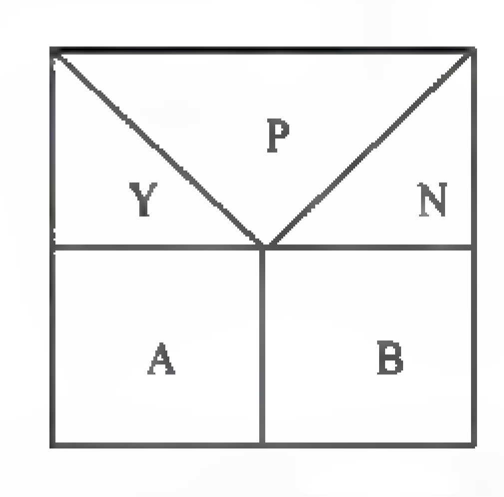


图 2.13　选择结构

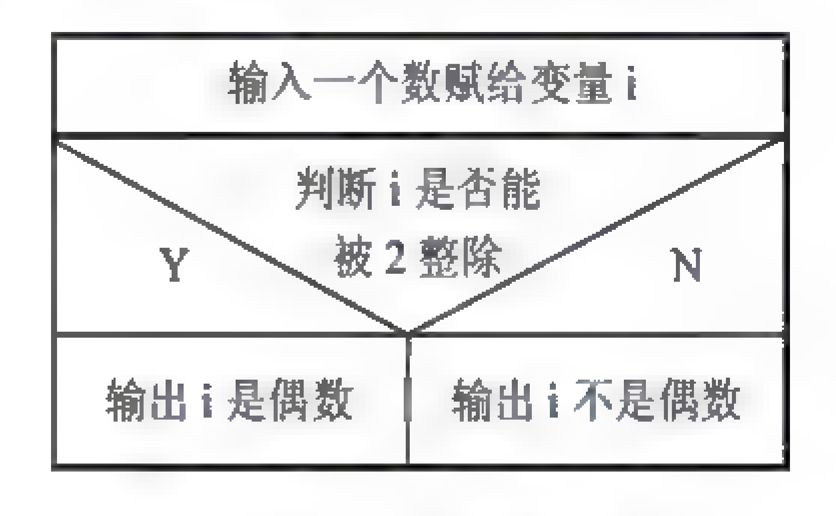


图 2.14　判断偶数

3. 循环结构

（1）当型循环的 N-S 流程图如图 2.15 所示。例 2.5 的当型循环的 N-S 流程图如图 2.16 所示。

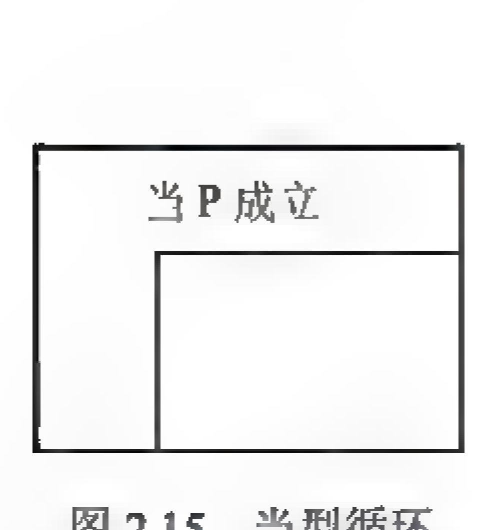


图 2.15　当型循环

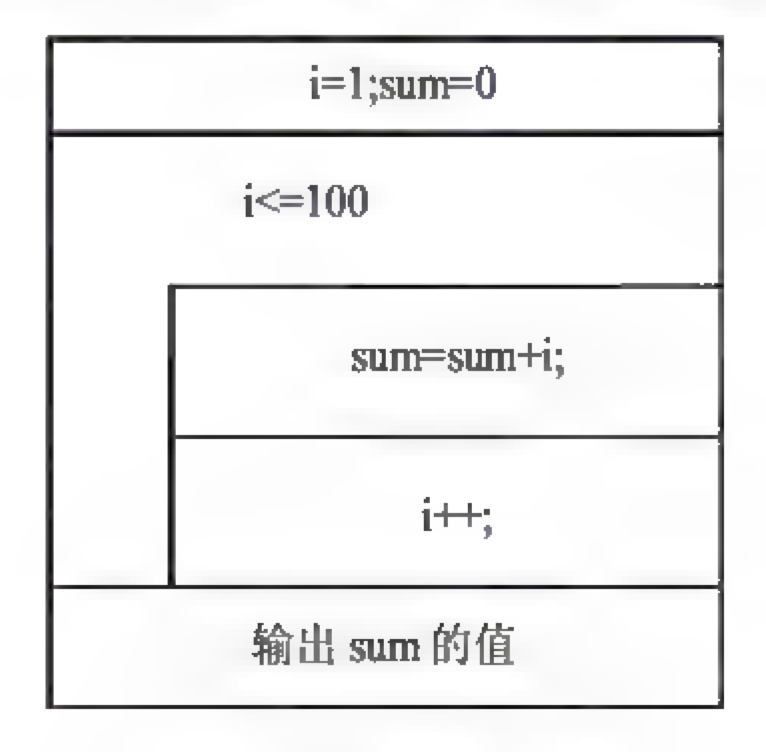


图 2.16　当型循环求和

（2）直到型循环的 N-S 图如图 2.17 所示。例 2.5 的直到型循环的 N-S 流程图如图 2.18 所示。

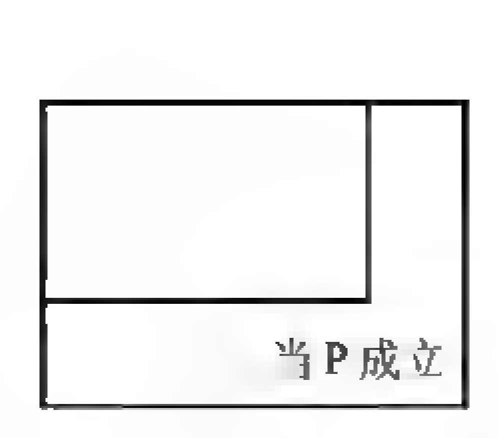


图 2.17　直到型循环

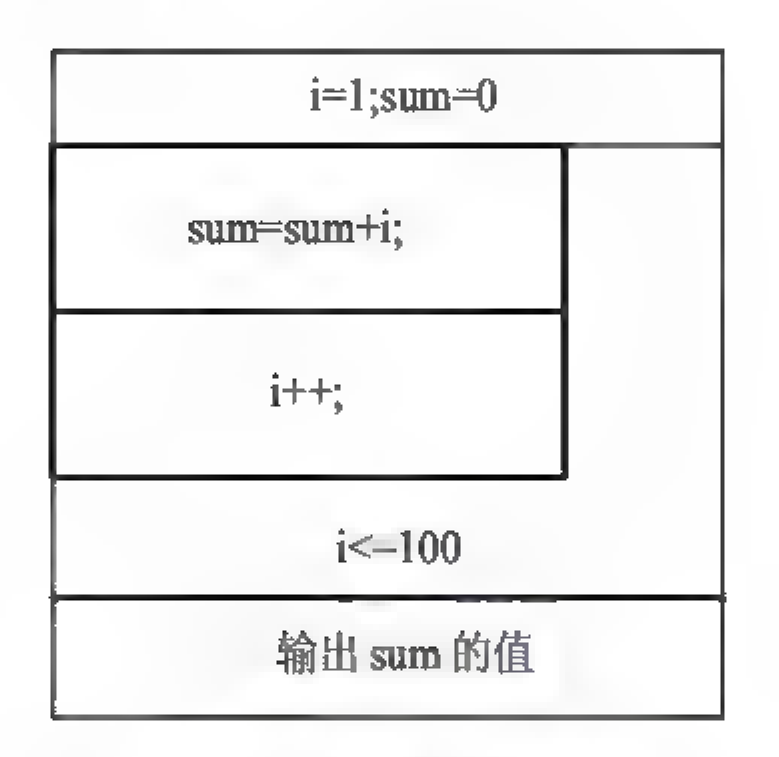


图 2.18　直到型循环求和



说明

这 3 种基本结构都只有一个入口和一个出口，结构内的每一部分都有可能被执行，且不会出现无终止循环的情况。

【例 2.6】　从键盘中输入一个数 n，求 n!。

本实例的流程图如图 2.19 所示。本实例的 N-S 流程图如图 2.20 所示。

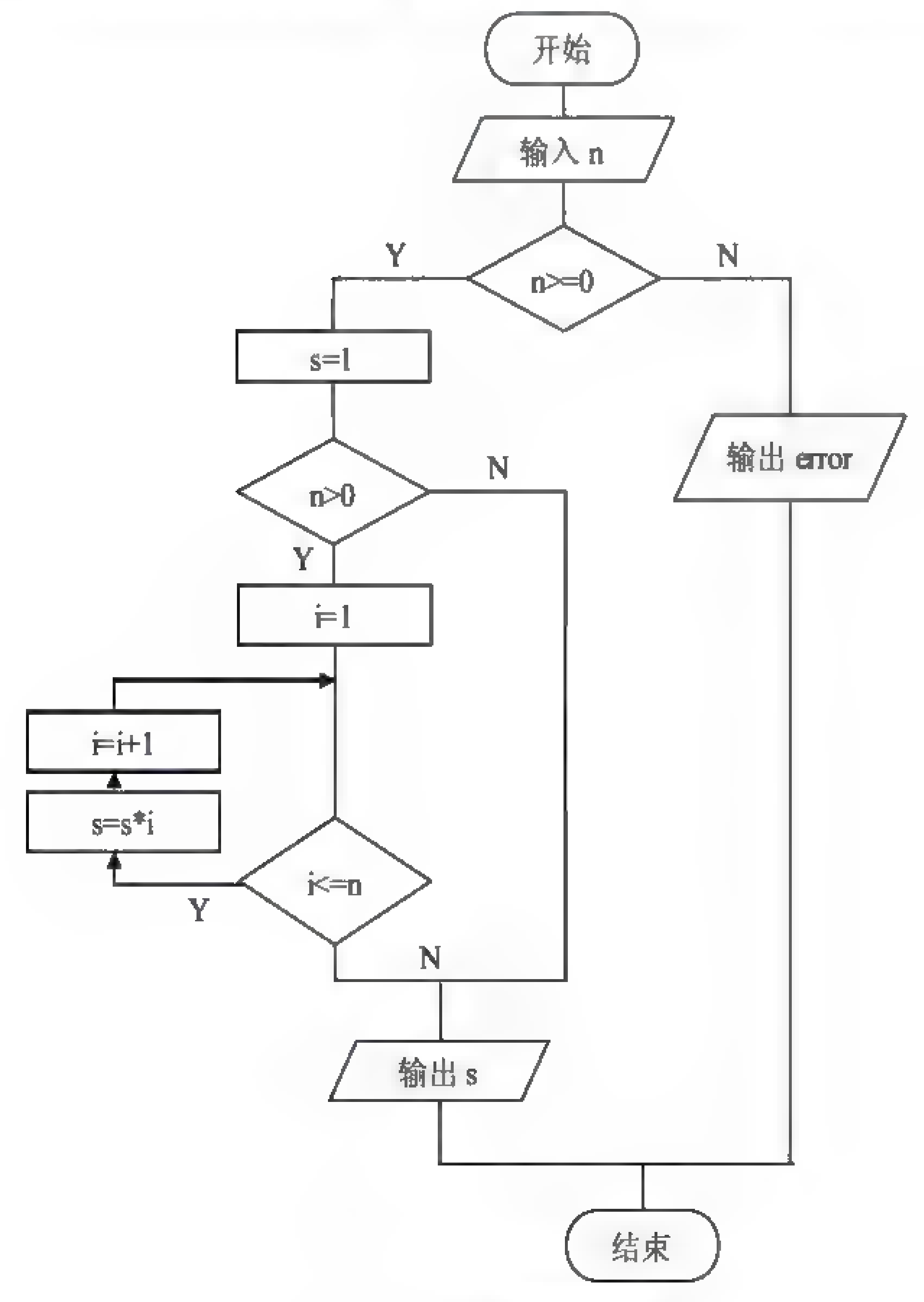


图 2.19　求 n!的流程图

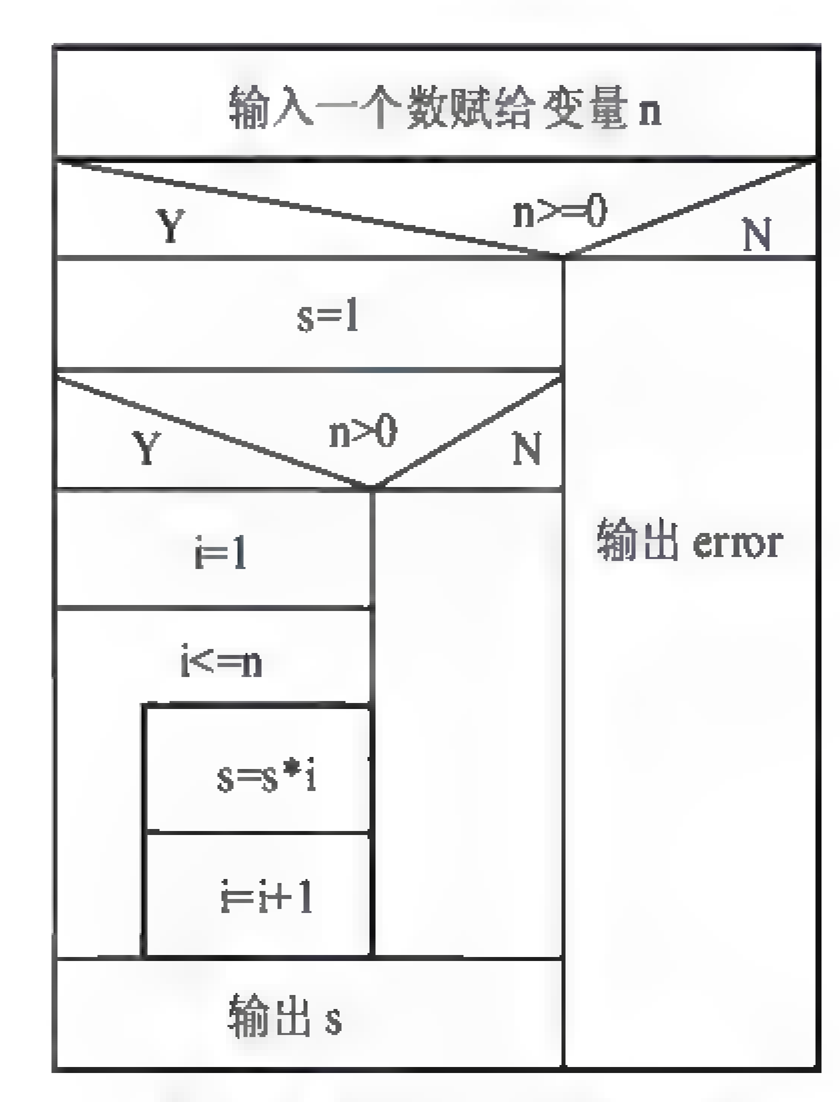


图 2.20　求 n!的 N-S 流程图

【例 2.7】　求两个数 a 和 b 的最大公约数。

本实例的流程图如图 2.21 所示。本实例的 N-S 流程图如图 2.22 所示。

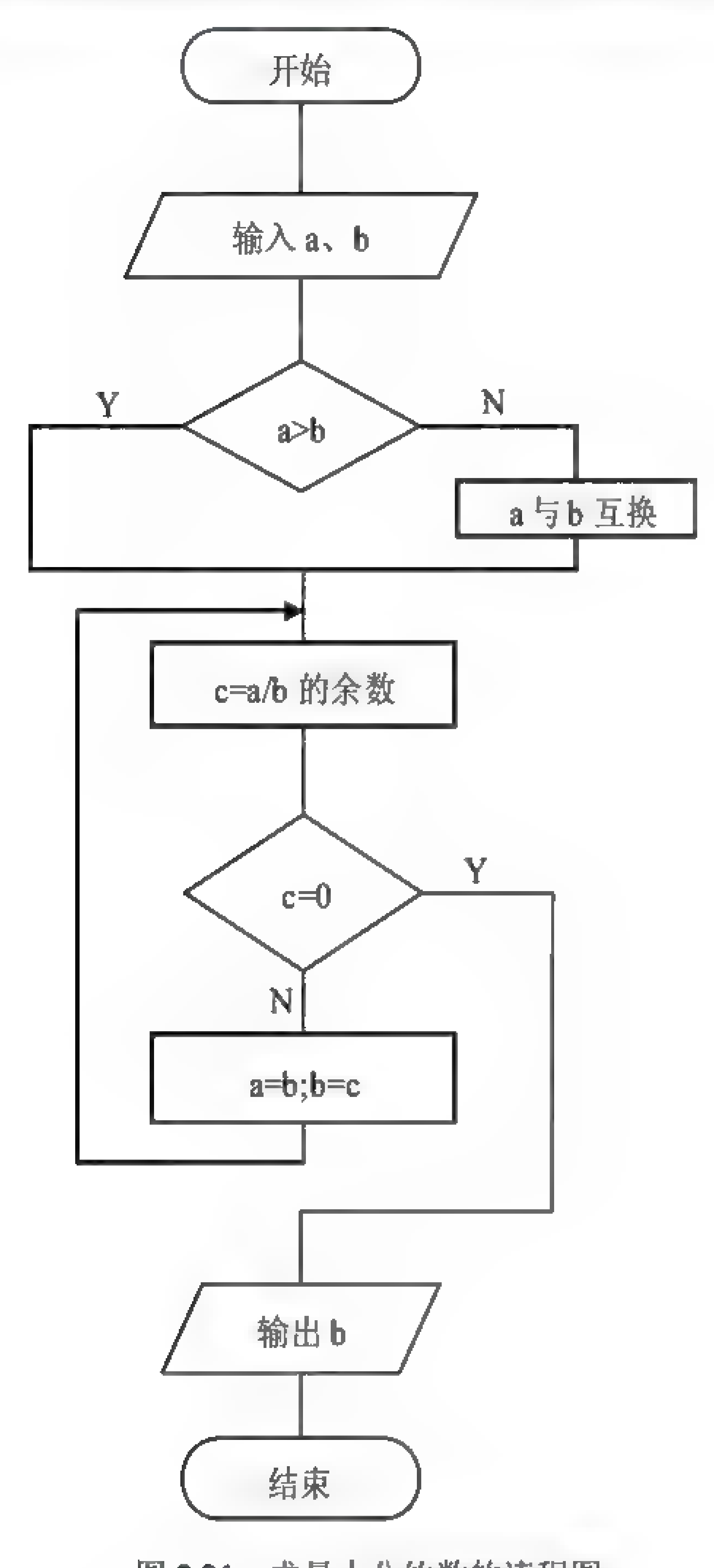


图 2.21　求最大公约数的流程图

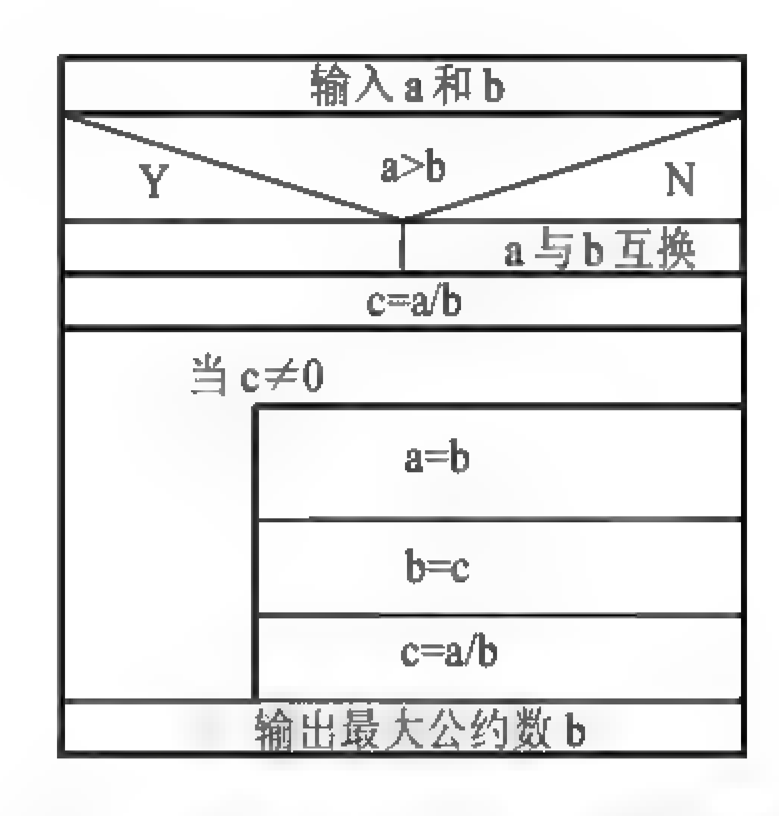


图 2.22　求最大公约数的 N-S 流程图

2.3　小　　结

本章主要介绍了算法的基本概念及算法描述两个方面的内容。算法的基本概念包括算法的特征和如何评价一个算法的优劣，算法的特征包括有穷性、确定性、可行性、输入和输出 5 个方面的内容，评价一个算法的优劣可从正确性、可读性、健壮性以及时间复杂度与空间复杂度这 4 个方面来考虑。算法描述介绍了自然语言、流程图和 N-S 流程图 3 种方法，其中要重点掌握顺序结构、选择结构和循环结构这 3 种基本结构的画法。

第 3 章

数据类型

（视频讲解：39 分钟）

在程序语言中，C 语言是十分重要的。学好了 C 语言，会很容易掌握其他编程语言，因为每种语言都会有一些共性存在。同时，一个好的程序员在编写代码时，一定要有规范性，清晰、整洁的代码才是有价值的。

本章致力于使读者掌握 C 语言中非常重要的一部分知识，即常量与变量。只有学懂了这些知识，才可以着手编写程序。

通过阅读本章，您可以：

- 了解编程规范的重要性
- 掌握如何使用常量
- 掌握变量在程序编写中的作用及重要性
- 区分变量的各种存储类别



3.1 编程规范

俗话说，“没有规矩，不成方圆。”虽然在 C 语言中编写代码是自由的，但是为了使编写的代码具有通用、友好的可读性，应该尽量按照一定的规范编写所设计的程序。

1. 代码缩进

代码统一缩进为 4 个字符。不采用空格，而用 Tab 键制表位。

```
#include<stdio.h>
int main()                              /*主函数 main*/
{
    int iResult=0;                      /*定义变量*/
    int i;
    printf("由 1 加到 100 的结果是：");     /*输出语句*/
    for(i=1;i<=100;i++)
    {
        iResult=i+iResult;
    }
    printf("%d\n",iResult);             /*输出结果*/
    return 0;                           /*结束返回*/
}
```

进行代码缩进

2. 变量、常量命名规范

常量命名统一为大写格式。如果是成员变量，均以 m_开始。如果是普通变量，取与实际意义相关的名称，要在前面添加类型的首字母，并且名称的首字母要大写。如果是指针，则在其标识符前添加 p 字符，并且名称首字母要大写。例如：

```
#define AGE    20                       /*定义常量*/
int m_iAge;                             /*定义整型成员变量*/
int iNumber;                            /*定义普通整型变量*/
int * pAge;                             /*定义指针变量*/
```

3. 函数的命名规范

在定义函数时，函数名的首字母要大写，其后的字母大小写混合。例如：

```
int AddTwoNum(int num1,int num2);
```

4. 注释

尽量采用行注释。如果行注释与代码处于一行，则注释应位于代码右方。如果连续出现多个行注释，并且代码较短，则应对齐注释。例如：

```
int iLong;                              /*长度*/
int iWidth;                             /*宽度*/
int iHeight                             /*高度*/
```



3.2　关　键　字

C 语言中有 32 个关键字，如表 3.1 所示。在今后的学习中将会逐渐接触到这些关键字的具体使用方法。

表 3.1　C 语言中的关键字

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	union	return
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	while	static	if



在 C 语言中，关键字是不允许作为标识符出现在程序中的。



3.3　标　识　符

在 C 语言中为了在程序的运行过程中可以使用变量、常量、函数、数组等，就要为这些形式设定一个名称，而设定的名称就是所谓的标识符。

外国人的姓名一般将名字放在前面，将家族的姓氏放在后面。在中国却恰恰相反，是把姓氏放在前面，将名字放在后面。从中可以看出，名字是可以随便起的，但也应该按照一定的要求进行设定。在 C 语言中，设定一个标识符的名称是非常自由的，可以设定自己喜欢、容易理解的名字，但这并不意味着可以任意自由发挥。下面介绍一下设定 C 语言标识符时应该遵守的一些命名规则。

（1）所有标识符必须由字母或下画线开头，而不能以数字或者符号开头。下面来看一些标识符命名示例：

```
int !number;                    /*错误，标识符第一个字符不能为符号*/
int 2hao;                       /*错误，标识符第一个字符不能为数字*/

int number;                     /*正确，标识符第一个字符为字母*/
int _hao;                       /*正确，标识符第一个字符为下画线*/
```

（2）在设定标识符时，除开头外，其他位置都可以由字母、下画线或数字组成。

☑ 在标识符中，有下画线的情况：

```
int good_way;                   /*正确，标识符中可以有下画线*/
```

☑ 在标识符中，有数字的情况：

```
int bus7;                        /*正确，标识符中可以有数字*/
int car6V;                       /*正确*/
```

（3）英文字母的大小写代表着不同的标识符。也就是说，在 C 语言中是区分大小写字母的。下面是一些正确的标识符：

```
int mingri;                      /*全部是小写*/
int MINGRI;                      /*全部是大写*/
int MingRi;                      /*一部分是小写，一部分是大写*/
```

从这些列出的标识符可以看出，只要标识符中的字符有一项是不同的，其代表的就是一个新的名称。

（4）标识符不能是关键字。关键字是定义一种类型使用的字符，标识符不能使用。例如，定义整型时系统使用了 int 关键字，因此用户定义的标识符就不能再使用 int，会提示编译错误。但将其中标识符的字母改写成大写字母，就可以通过编译。

```
int int;                         /*错误！*/
int Int;                         /*正确，改变标识符中的字母为大写*/
```

（5）标识符的命名最好具有相关的含义。将标识符设定成有一定含义的名称，不但可以方便程序编写，并且在以后回顾程序时，或者他人阅读自己的程序时，会更容易读懂。例如，在定义一个长方体的长、宽和高时，只图一时的方便可以简单地进行定义：

```
int a;                           /*代表长度*/
int b;                           /*代表宽度*/
int c;                           /*代表高度*/
int iLong;
int iWidth;
int iHeight;
```

从上面列举出的标识符可以看出，标识符的设定如果不具有一定的含义，没有后面的注释就很难使人理解要代表的作用是什么。如果将标识符设定得具有其功能含义，那么通过直观地查看就可以了解其具体的作用功能。

（6）ANSI 标准规定，标识符可以为任意长度，但外部名必须仅前 6 个字符就能唯一地进行区分。这是因为某些编译程序（如 IBM PC 的 MS C）仅能识别前 6 个字符。



3.4 数据类型

程序在运行时要做的内容就是处理数据。程序要解决复杂的问题，就要处理不同的数据。不同的数据都是以自己本身的一种特定形式存在的（如整型、字符型、实型等），不同的数据类型占用不同的存储空间。C 语言中有多种不同的数据类型，其中包括基本类型、构造类型、指针类型和空类型等。这里先通过图 3.1 看一下其组织结构，然后再对每一种类型进行相应的讲解。

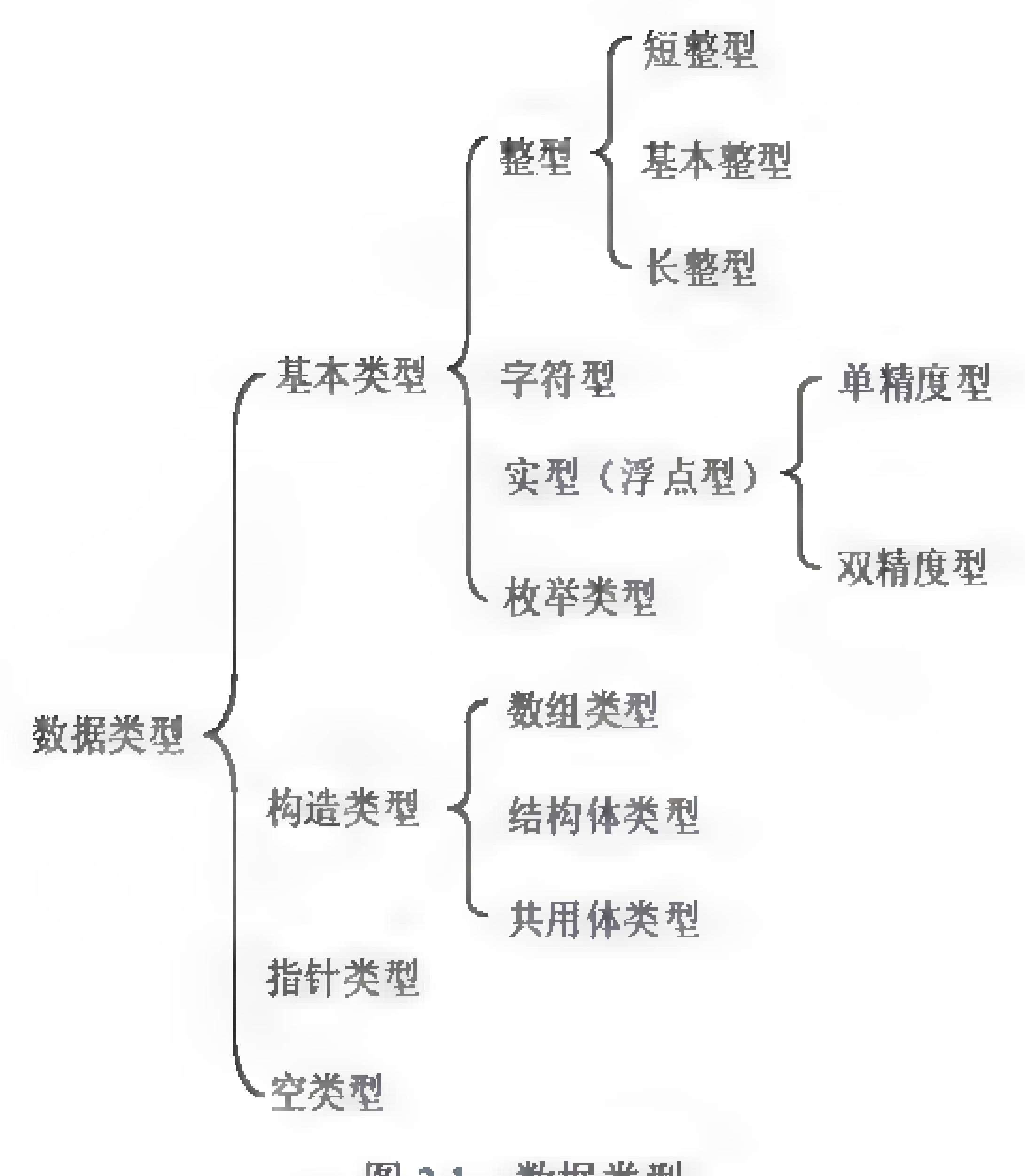


图 3.1 数据类型

1. 基本类型

基本类型也就是C语言中的基础类型，其中包括整型、字符型、实型（浮点型）、枚举类型。

2. 构造类型

构造类型就是使用基本类型的数据，或者使用已经构造好的数据类型，进行添加、设计，构造出新的数据类型，使新构造的类型能满足待解决问题所需要的数据类型。

通过构造类型的说明可以看出，它并不像基本类型那样简单，而是由多种类型组合而成的新类型，其中每一组成部分称为构造类型的成员。构造类型包括数组类型、结构体类型和共用体类型3种形式。

3. 指针类型

C语言的精华是什么？是指针！指针类型不同于其他类型，因为其值表示的是某个内存地址。

4. 空类型

空类型的关键字是void，其主要作用在于如下两点：

☑ 对函数返回的限定。

☑ 对函数参数的限定。

也就是说，一般一个函数都具有一个返回值，将其值返回调用者。这个返回值应该是具有特定的类型，如整型int。但是当函数不必返回一个值时，就可以使用空类型设定返回值的类型。

3.5 常 量



在介绍常量之前，先来了解一下什么是常量。常量就是其值在程序运行过程中不可以改变的量。常量可以分为以下三大类：

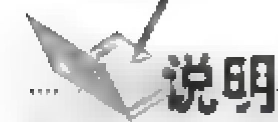
☑ 数值型常量。包括整型常量和实型常量。
☑ 字符型常量。
☑ 符号常量。

下面将对常量进行详细的说明。

3.5.1 整型常量

整型常量就是直接使用的整型常数，如 123、 456 等。整型常量可以是长整型、短整型、符号整型和无符号整型。

☑ 无符号短整型的取值范围是 0～65535，有符号短整型的取值范围是 32768～+32767，这些都是 16 位整型常量的范围。
☑ 如果整型是 32 位的，那么无符号形式的取值范围是 0～4294967295，有符号形式的取值范围是-2147483648～+2147483647。但是整型如果是 16 位的，就与无符号短整型的范围相同。



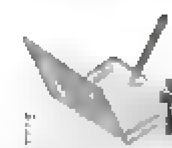
说明

不同的编译器，整型的取值范围是不一样的。在字长为 16 位的计算机中，整型就为 16 位；在字长为 32 位的计算机中，整型就为 32 位。

☑ 长整型是 32 位的，其取值范围可以参考上面有关整型的描述。

在编写整型常量时，可以在常量的后面加上符号 L 或者 U 进行修饰。L 表示该常量是长整型，U 表示该常量为无符号整型，例如：

```
LongNum= 1000L;                    /*L 表示长整型*/
UnsignLongNum=500U;                /*U 表示无符号整型*/
```



说明

表示长整型和无符号整型的后缀字母 L 和 U 可以使用大写，也可以使用小写。

整型常量可以通过 3 种形式进行表达，分别为八进制形式、十进制形式和十六进制形式。下面分别进行介绍。

1．八进制整数

要使得使用的数据表达形式是八进制，需要在常数前加上 0 进行修饰。八进制所包含的数字是 0～7。例如：

```
OctalNumber1=0123;                 /*在常数前面加上一个 0 来代表八进制*/
OctalNumber2=0432;
```



注意

以下关于八进制的写法是错误的：

```
OctalNumber3=356;                  /*没有前缀 0*/
OctalNumber4=0492;                 /*包含了非八进制数 9*/
```

2. 十进制整数

十进制是不需要在其前面添加前缀的。十进制中所包含的数字为 0～9。例如：

```
AlgorismNumber1=123;
AlgorismNumber2=456;
```

这些整型数据都以二进制的方式存放在计算机的内存之中，其数值是以补码的形式进行表示的。一个正数的补码与其原码形式相同，一个负数的补码是将该数绝对值的二进制形式按位取反后再加 1。例如，一个十进制数 11 在内存中的表现形式如图 3.2 所示。

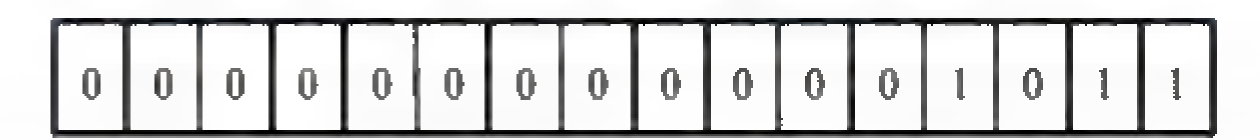


图 3.2　十进制数 11 在内存中的表现形式

如果是-11，那么在内存中又是怎样的呢？因为是以补码进行表示，所以负数要先将其绝对值求出，如图 3.2 所示；然后进行取反操作，如图 3.3 所示，得到取反后的结果。

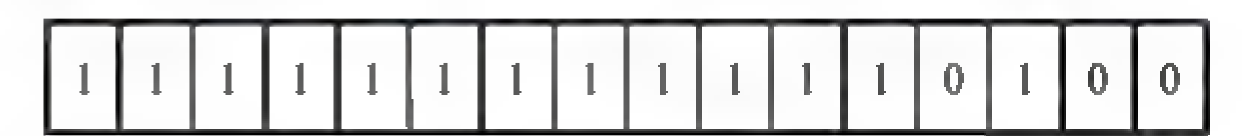


图 3.3　进行取反操作

取反之后还要进行加 1 操作，这样就得到最终的结果。如图 3.4 所示为-11 在计算机内存中的存储情况。

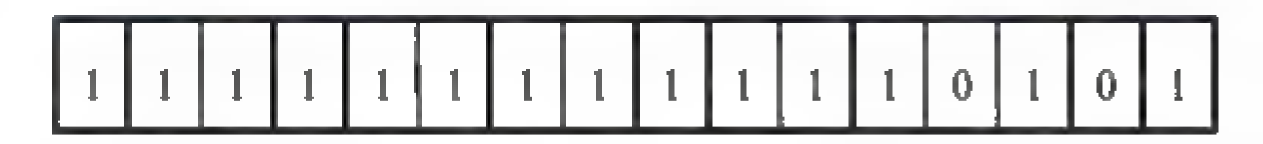
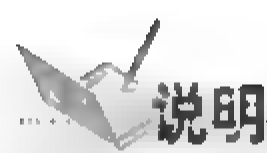


图 3.4　加 1 操作



说明

对于有符号整数，其在内存中存放的最左面一位表示符号位，如果该位为 0，则说明该数为正；若为 1，则说明该数为负。

3. 十六进制整数

常量前面使用 0x 作为前缀，表示该常量是用十六进制进行表示的。十六进制中包含数字 0～9 以及字母 A～F。例如：

```
HexNumber1=0x123;                /*加上前缀 0x 表示常量为十六进制*/
HexNumber2=0x3ba4;
```



说明

其中字母 A～F 可以使用大写形式，也可以使用 a～f 小写形式。



注意

以下关于十六进制的写法是错误的：

```
HexNumber1=123;            /*没有前缀 0x*/
HexNumber2=0x89j2;         /*包含了非十六进制的字母 j*/
```

技巧

Windows 操作系统中，在“开始”菜单的“附件”命令中有一个计算器软件，可以使用这个小软件进行八进制、十进制和十六进制之间的转换。这里需要注意的是，要选用科学型计算器，如图 3.5 所示。调用方法是在其“查看”菜单中选择“科学型”命令。

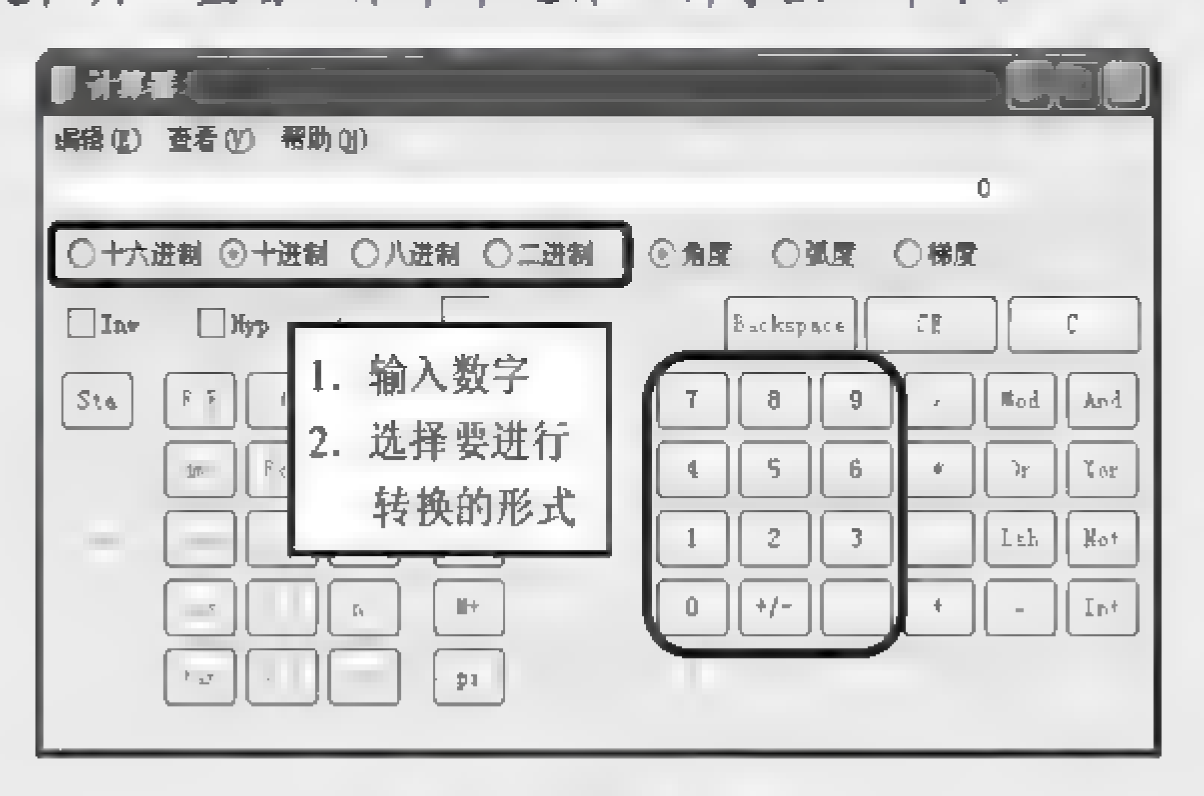


图 3.5　科学型计算器

3.5.2　实型常量

实型也称为浮点型，由整数部分和小数部分组成，并用十进制的小数点进行分隔。表示实数的方式有科学计数方式和指数方式两种。

1. 科学计数方式

科学计数方式就是使用十进制的小数方法描述实型，例如：

```
SciNum1=123.45;                    /*科学计数法*/
SciNum2=0.5458;
```

2. 指数方式

有时，实型非常大或者非常小，使用科学计数方式是不利于观察的。这时，可以使用指数方法显示实型常量。其中，使用字母 e 或者 E 进行指数显示，如 45e2 表示的就是 4500，而 45e-2 表示的就是 0.45。如上面的 SciNum1 和 SciNum2 代表的实型常量，使用指数方式显示这两个实型常量，代码如下：

```
SciNum1=1.2345e2;                  /*指数方式显示*/
SciNum2=5.458e-1;
```

在编写实型常量时，可以在常量的后面加上符号 F 或者 L 进行修饰。F 表示该常量是 float 单精度类型，L 表示该常量为 long double 长双精度类型。例如：

```
FloatNum= 1.2345e2F;               /*单精度类型*/
LongDoubleNum=5.458e-1L;           /*长双精度类型*/
```

如果不在后面加上后缀，那么在默认状态下，实型常量为double双精度类型。例如：

```
DoubleNum= 1.2345e2;                    /*双精度类型*/
```

注意

后缀的大小写是通用的。

3.5.3 字符型常量

字符型常量与之前所介绍的常量有所不同，即要对其字符型常量使用指定的定界符进行限制。字符型常量可以分成两种：一种是字符常量，另一种是字符串常量。下面分别对这两种字符型常量进行介绍。

1．字符常量

使用单直撇括起一个字符，这种形式就是字符常量。例如，'A'、'#'、'b'等都是正确的字符常量。在这里，需要注意以下几点：

☑ 字符常量只能包括一个字符，不能是字符串。例如，'A'是正确的，但是用'AB'来表示字符常量就是错误的。

☑ 字符常量是区分大小写的。例如，'A'字符和'a'字符是不一样的，这两个字符代表着不同的字符常量。

☑ ' '这对单直撇代表着定界符，不属于字符常量中的一部分。

【例3.1】 字符常量的输出。（**实例位置：资源包\TM\sl\3\1**）

在本实例中，使用putchar函数将单个字符常量进行输出，使得输出的字符常量形成一个单词Hello并显示在控制台中。

```
#include<stdio.h>
int main()
{
    putchar('H');                       /*输出字符常量H*/
    putchar('e');                       /*输出字符常量e*/
    putchar('l');                       /*输出字符常量l*/
    putchar('l');                       /*输出字符常量l*/
    putchar('o');                       /*输出字符常量o*/
    putchar('\n');                      /*进行换行*/
    return 0;
}
```

运行程序，显示效果如图3.6所示。

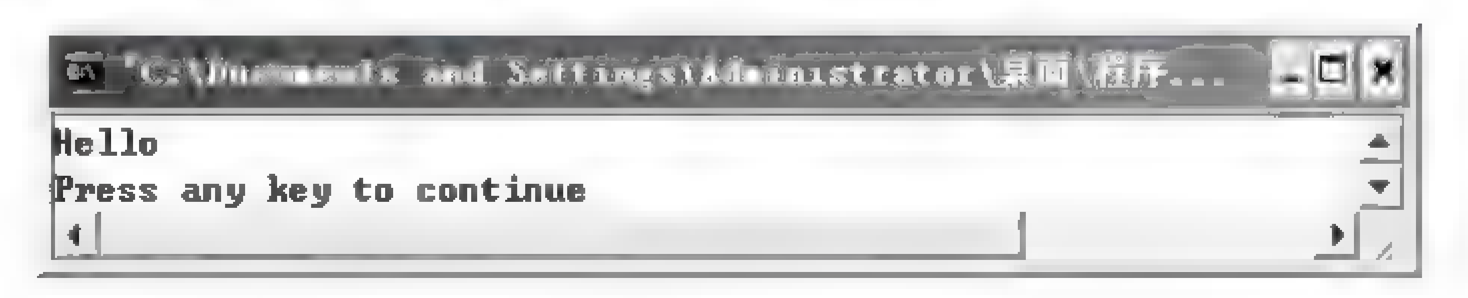


图3.6 使用字符常量

2. 字符串常量

字符串常量是用一组双引号括起来的若干字符序列，例如，“Have a good day!”和“beautiful day”即为字符串常量。如果字符串中一个字符都没有，将其称作空串，此时字符串的长度为 0。

在 C 语言中存储字符串常量时，系统会在字符串的末尾自动加一个“\0”，作为字符串的结束标志。例如，字符串“welcome”在内存中的存储形式如图 3.7 所示。



w	e	l	c	o	m	e	\0

图 3.7 “\0”为系统所加

注意

在程序中编写字符串常量时，不必在字符串的结尾处加上“\0”结束字符，系统会自动添加结束字符。

【例 3.2】 输出字符串常量。（**实例位置：资源包\TM\sl\3\2**）

在本实例中，使用 printf 函数将字符串常量“What a nice day!”在控制台输出显示。

```
#include<stdio.h>                          /*包含头文件*/

int main()
{
    printf("What a nice day!\n");          /*输出字符串*/
    return 0;                              /*程序结束*/
}
```

运行程序，显示效果如图 3.8 所示。

上面介绍了有关字符常量和字符串常量的内容，那么同样是字符，它们之间有什么差别呢？其差别主要体现在以下 3 个方面：

☑ 定界符的使用不同。字符常量使用的是单直撇，而字符串常量使用的是双引号。

☑ 长度不同。上面提到过字符常量只能有一个字符，也就是说字符常量的长度就是 1。字符串常量的长度可以是 0，也可以是任意数。即使字符串常量中的字符数量只有 1 个，长度也不是 1。例如，字符串常量“H”，其长度为 2。通过图 3.9 可以体会到，字符串常量“H”的长度为 2 的原因。



图 3.8 输出字符串

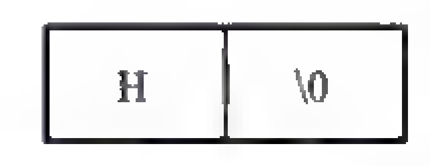


图 3.9 字符串“H”



说明

还记得在字符串常量中有关结束字符的介绍吗？系统会自动在字符串的尾部添加一个字符串结束字符“\0”，这也就是“H”的长度是 2 的原因。

☑ 存储的方式不同，在字符常量中存储的是字符的 ASCII 码值；而在字符串常量中，不仅要存储有效的字符，还要存储结尾处的结束标志“\0”。

前面提到过有关 ASCII 码的内容，那么 ASCII 是什么呢？在 C 语言中，所使用的字符被一一映射到一个表中，这个表称为 ASCII 码表，如表 3.2 所示。

表 3.2 ASCII 表

ASCII 值	缩写/字符	解　释
0	NUL（null）	空字符（\0）
1	SOH（star to fhanding）	标题开始
2	STX（star to ftext）	正文开始
3	ETX（end of text）	正文结束
4	EOT（end of transmission）	传输结束
5	ENQ（enquiry）	请求
6	ACK（acknowledge）	收到通知
7	BEL（bell）	响铃（\a）
8	BS（backspace）	退格（\b）
9	HT（horizontal tab）	水平制表符（\t）
10	LF（NL）（linefeed,newline）	换行键（\n）
11	VT（verticaltab）	垂直制表符
12	FF（NP）（formfeed,newpage）	换页键（\f）
13	CR（carriagereturn）	回车键（\r）
14	SO（shift out）	不用切换
15	SI（shift in）	启用切换
16	DLE（data link escape）	数据链路转义
17	DC1（device control1）	设备控制 1
18	DC2（device control2）	设备控制 2
19	DC3（device control3）	设备控制 3
20	DC4（device control4）	设备控制 4
21	NAK（negative acknowledge）	拒绝接收
22	SYN（synchronousidle）	同步空闲
23	ETB（end of trans.block）	传输块结束
24	CAN（cancel）	取消
25	EM（end of medium）	介质中断
26	SUB（substitute）	替补
27	ESC（escape）	溢出
28	FS（file separator）	文件分割符
29	GS（group separator）	分组符
30	RS（record separator）	记录分离符
31	US（unit separator）	单元分隔符
32	……	完整表参见附录 A

3.5.4　转义字符

在前面的例 3.1 和例 3.2 中都能看到“\n”符号，输出结果中却不显示该符号，只是进行了换行操作，这种符号称为转义符号。

转义符号在字符常量中是一种特殊的字符。转义字符是以反斜杠“\”开头的字符，后面跟一个或几个字符。常用的转义字符及其含义如表 3.3 所示。

表 3.3　常用的转义字符表

转 义 字 符	意　　义	转 义 字 符	意　　义
\n	回车换行	\\	反斜杠“\”
\t	横向跳到下一制表位置	\'	单引号符
\v	竖向跳格	\a	鸣铃
\b	退格	\ddd	1～3 位八进制数所代表的字符
\r	回车	\xhh	1～2 位十六进制数所代表的字符
\f	走纸换页		

3.5.5　符号常量

在例 1.2 中，程序的功能是求解一个长方体的体积，其中长方体的高度是固定的，使用一个符号名代替固定的常量值，这里使用的符号名就称之为符号常量。使用符号常量的好处在于可以为编程和阅读带来方便。

【例 3.3】　符号常量的使用。（**实例位置：资源包\TM\sl\3\3**）

本实例使用符号常量来表示圆周率，在控制台上显示文字提示用户输入数据，该数据是有关圆半径的值。得到用户输入的半径，经过计算得到圆的面积，最后将结果显示出来。

```
#include<stdio.h>
#define PAI 3.14                              /*定义符号常量*/

int main()
{
    double fRadius;                           /*定义半径变量*/
    double fResult=0;                         /*定义结果变量*/
    printf("请输入圆的半径：");                  /*提示*/
    scanf("%lf",&fRadius);                    /*输入数据*/
    fResult=fRadius*fRadius*PAI;              /*进行计算*/
    printf("圆的面积为：%lf\n",fResult);        /*显示结果*/
    return 0;                                 /*程序结束*/
}
```

运行程序，显示效果如图 3.10 所示。

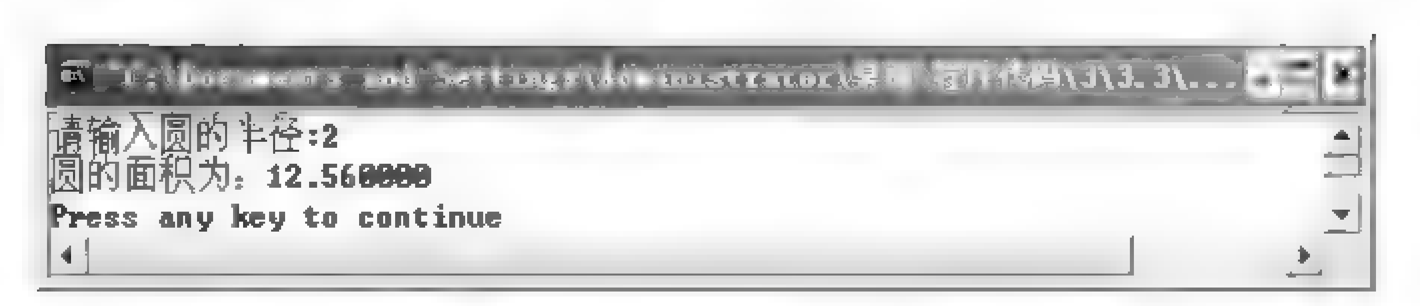


图 3.10 符号常量的使用

3.6 变　　量



在前面的例子中已经多次接触过变量。变量就是在程序运行期间其值可以变化的量。每个变量都属于一种类型，每种类型都定义了变量的格式和行为。因此，一个变量应该有属于自己的名称，并且在内存中占有存储空间，其中，变量的大小取决于类型。C 语言中的变量类型包括整型变量、实型变量和字符型变量。

3.6.1 整型变量

整型变量是用来存储整型数值的变量。整型变量可以分为如表 3.4 所示的 6 种类型，其中基本类型的符号使用 int 关键字，在此基础上可以根据需要加上一些符号进行修饰，如关键字 short 或 long。

表 3.4 整型变量的分类

类　型	关 键 字
有符号基本整型	[signed] int
无符号基本整型	unsigned [int]
有符号短整型	[signed] short [int]
无符号短整型	unsigned short [int]
有符号长整型	[signed] long [int]
无符号长整型	unsigned long [int]


说明

表格中的[]为可选部分。例如[signed] int，在编写时可以省略 signed 关键字。

1. 有符号基本整型

有符号基本整型是指 signed int 型，其值是基本的整型常数。编写时，常将其关键字 signed 进行省略。有符号基本整型在内存中占 4 个字节，取值范围是-2147483648～2147483647。


说明

通常说到的整型，都是指有符号基本整型 int。

定义一个有符号整型变量的方法是在变量前使用关键字 int。例如，定义一个整型的变量 iNumber，并为其赋值 10 的方法如下：

```
int iNumber;                                  /*定义有符号基本整型变量*/
iNumber=10;                                   /*为变量赋值*/
```

或者在定义变量的同时为其赋值：

```
int iNumber=10;                               /*定义有符号基本整型变量*/
```

注意

在编写程序时，程序中用到的所有变量的定义应该放在变量的赋值之前，否则会产生错误。通过下面的两个例子进行对比：

```
/*正确的写法：*/
int iNumber1;                                 /*先定义变量*/
int iNumber2;
iNumber1=6;                                   /*再对变量赋值*/
iNumber2=7;

/*错误的写法：*/
int iNumber1;                                 /*定义变量*/
iNumber1=6;                                   /*为变量赋值，错误！！因为赋值语句在定义变量语句之前*/
int iNumber2;                                 /*定义变量*/
iNumber2=7;                                   /*为变量赋值*/
```

【例 3.4】 有符号基本整型。（**实例位置：资源包\TM\sl\3\4**）

本实例是对有符号基本整型变量的使用，可使读者更为直观地看到其作用。

```
#include<stdio.h>
int main()
{
    signed int iNumber;                       /*定义有符号基本整型变量*/
    iNumber=10;                               /*为变量进行赋值*/
    printf("%d\n",iNumber);                   /*显示整型变量*/
    return 0;                                 /*程序结束*/
}
```

运行程序，显示效果如图 3.11 所示。



图 3.11　有符号基本整型

2．无符号基本整型

无符号基本整型使用的关键字是 unsigned int，其中的关键字 int 在编写时可以省略。无符号基本整

型在内存中占4个字节，取值范围是0～4294967295。

定义一个无符号基本整型变量的方法是在变量前使用关键字unsigned。例如，要定义一个无符号基本整型的变量iUnsignedNum，并为其赋值10的方法如下：

```
unsigned iUnsignedNum;                /*定义无符号基本整型变量*/
iUnsignedNum=10;                      /*为变量赋值*/
```

3．有符号短整型

有符号短整型使用的关键字是signed short int，其中的关键字signed和int在编写时可以省略。有符号短整型在内存中占两个字节，取值范围是-32768～32767。

定义一个有符号短整型变量的方法是在变量前使用关键字short。例如，要定义一个有符号短整型的变量iShortNum，并为其赋值10的方法如下：

```
short iShortNum;                      /*定义有符号短整型变量*/
iShortNum=10;                         /*为变量赋值*/
```

4．无符号短整型

无符号短整型使用的关键字是unsigned short int，其中的关键字int在编写时可以省略。无符号短整型在内存中占两个字节，取值范围是0～65535。

定义一个无符号短整型变量的方法是在变量前使用关键字unsigned short。例如，要定义一个无符号短整型的变量iUnsignedShtNum，并为其赋值10的方法如下：

```
unsigned short iUnsignedShtNum;       /*定义无符号短整型变量*/
iUnsignedShtNum=10;                   /*为变量赋值*/
```

5．有符号长整型

有符号长整型使用的关键字是signed long int，其中的关键字signed和int在编写时可以省略。有符号长整型在内存中占4个字节，取值范围是-2147483648～2147483647。

定义一个有符号长整型变量的方法是在变量前使用关键字long。例如，要定义一个有符号长整型的变量iLongNum，并为其赋值10的方法如下：

```
long iLongNum;                        /*定义有符号长整型变量*/
iLongNum=10;                          /*为变量赋值*/
```

6．无符号长整型

无符号长整型使用的关键字是unsigned long int，其中的关键字int在编写时可以省略。无符号长整型在内存中占4个字节，取值范围是0～4294967295。

定义一个无符号长整型变量的方法是在变量前使用关键字unsigned long。例如，要定义一个无符号长整型的变量iUnsignedLongNum，并为其赋值10的方法如下：

```
unsigned long iUnsignedLongNum;       /*定义无符号长整型变量*/
iUnsignedLongNum=10;                  /*为变量赋值*/
```

3.6.2 实型变量

实型变量也称为浮点型变量，是指用来存储实型数值的变量，其中实型数值是由整数和小数两部分组成的。实型变量根据实型的精度可以分为单精度类型、双精度类型和长双精度类型 3 种，如表 3.5 所示。

表 3.5 实型变量的分类

类 型 名 称	关 键 字
单精度类型	float
双精度类型	double
长双精度类型	long double

1. 单精度类型

单精度类型使用的关键字是 float，它在内存中占 4 个字节，取值范围是 $-3.4\times10^{-38}\sim3.4\times10^{38}$。

定义一个单精度类型变量的方法是在变量前使用关键字 float。例如，要定义一个变量 fFloatStyle，并为其赋值 3.14 的方法如下：

```
float fFloatStyle;                /*定义单精度类型变量*/
fFloatStyle=3.14f;                /*为变量赋值*/
```

【例 3.5】 使用单精度类型变量。（**实例位置：资源包\TM\sl\3\5**）

在本实例中，定义一个单精度类型变量，然后为其赋值 1.23，最后通过输出语句将其显示在控制台。

```
#include<stdio.h>

int main()
{
    float fFloatStyle;            /*定义单精度类型变量*/
    fFloatStyle=1.23f;            /*为变量赋值*/
    printf("%f\n",fFloatStyle);   /*输出变量的值*/
    return 0;                     /*程序结束*/
}
```

运行程序，显示效果如图 3.12 所示。



图 3.12 使用单精度类型变量

2. 双精度类型

双精度类型使用的关键字是 double，它在内存中占 8 个字节，取值范围是 $-1.7\times10^{-308}\sim1.7\times10^{308}$。

定义一个双精度类型变量的方法是在变量前使用关键字 double。例如，要定义一个变量 dDoubleStyle，并为其赋值 5.321 的方法如下：

```
double dDoubleStyle;                    /*定义双精度类型变量*/
dDoubleStyle=5.321;                     /*为变量赋值*/
```

【例 3.6】 使用双精度类型变量。（实例位置：资源包\TM\sl\3\6）

在本实例中，定义一个双精度类型变量，然后为其赋值 61.458，最后通过输出语句将其显示在控制台。

```
#include<stdio.h>

int main()
{
    double dDoubleStyle;                /*定义一个双精度类型变量*/
    dDoubleStyle=61.458;                /*为变量赋值*/
    printf("%f\n",dDoubleStyle);        /*显示变量值*/
    return 0;                           /*程序结束*/
}
```

运行程序，显示效果如图 3.13 所示。



图 3.13 使用双精度类型变量

3. 长双精度类型

长双精度类型使用的关键字是 long double，它在内存中占 8 个字节，取值范围是-1.7×10^{-308}～1.7×10^{308}。

定义一个双精度类型变量的方法是在变量前使用关键字 long double。例如，要定义一个变量 fLongDouble，并为其赋值 46.257 的方法如下：

```
long double fLongDouble;                /*定义长双精度类型变量*/
fLongDouble=46.257;                     /*为变量赋值*/
```

【例 3.7】 使用长双精度类型变量。（实例位置：资源包\TM\sl\3\7）

在本实例中，定义一个长双精度类型变量，然后为其赋值 46.257，最后通过输出语句将其显示在控制台。

```
#include<stdio.h>

int main()
{
    long double fLongDouble;            /*定义长双精度变量*/
    fLongDouble=46.257;                 /*为变量赋值*/
    printf("%f\n",fLongDouble);         /*将变量值进行输出*/
    return 0;                           /*程序结束*/
}
```

运行程序，显示效果如图 3.14 所示。

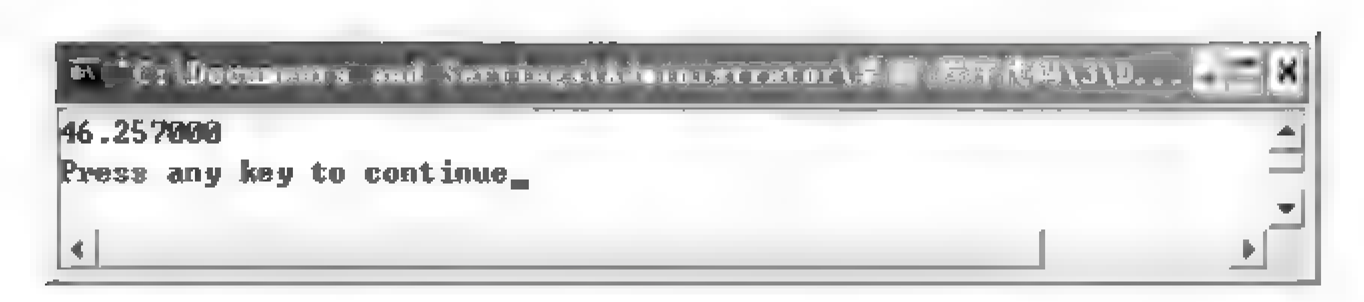


图 3.14　使用长双精度类型变量

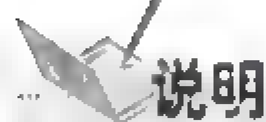
3.6.3　字符型变量

字符型变量是用来存储字符常量的变量。将一个字符常量存储到一个字符变量中，实际上是将该字符的 ASCII 码值（无符号整数）存储到内存单元中。

字符型变量在内存空间中占一个字节，取值范围是-128～127。

定义一个字符型变量的方法是使用关键字 char。例如，要定义一个字符型的变量 cChar，为其赋值为'a'的方法如下：

```
char cChar;                                    /*定义字符型变量*/
cChar= 'a';                                    /*为变量赋值*/
```


说明

字符数据在内存中存储的是字符的 ASCII 码，即一个无符号整数，其形式与整数的存储形式一样，因此 C 语言允许字符型数据与整型数据之间通用。例如：

```
char cChar1;                                   /*字符型变量 cChar1*/
char cChar2;                                   /*字符型变量 cChar2*/
cChar1='a';                                    /*为变量赋值*/
cChar2=97;

printf("%c\n",cChar1);                         /*显示结果为 a*/
printf("%c\n",cChar2);                         /*显示结果为 a*/
```

从上面的代码中可以看到，首先定义两个字符型变量，在为两个变量进行赋值时，一个变量赋值为'a'，而另一个赋值为 97。最后显示结果都是字符'a'。

【例 3.8】　使用字符型变量。（**实例位置：资源包\TM\sl\3\8**）

本实例为定义的字符型变量和整型变量进行不同的赋值，然后通过输出结果观察整型变量和字符型变量之间的转换。

```
#include<stdio.h>
int main()
{
    char cChar1;                               /*字符型变量 cChar1*/
    char cChar2;                               /*字符型变量 cChar2*/
    int iInt1;                                 /*整型变量 iInt1*/
    int iInt2;                                 /*整型变量 iInt2*/

    cChar1='a';                                /*为变量赋值*/
```

```
    cChar2=97;
    iInt1='a';
    iInt2=97;

    printf("%c\n",cChar1);                /*显示结果为 a*/
    printf("%d\n",cChar2);                /*显示结果为 97*/
    printf("%c\n",iInt1);                 /*显示结果为 a*/
    printf("%d\n",iInt2);                 /*显示结果为 97*/
    return 0;                             /*程序结束*/
}
```

运行程序，显示效果如图 3.15 所示。



图 3.15　使用字符型变量

以上就是有关整型变量、实型变量和字符型变量的相关知识。下面使用一个表格对这些知识进行一下概括总结，如表 3.6 所示。

表 3.6　数值型和字符型数据的字节数和数值范围

类　型	关 键 字	字　节	数 值 范 围
整型	[signed] int	4	-2147483648～2147483647
无符号整型	unsigned [int]	4	0～4294967295
短整型	short [int]	2	-32768～32767
无符号短整型	unsigned short [int]	2	0～65535
长整型	long [int]	4	-2147483648～2147483647
无符号长整型	unsigned long [int]	4	0～4294967295
字符型	[signed] char	1	-128～127
无符号字符型	unsigned char	1	0～255
单精度型	float	4	-3.4×10^{-38}～3.4×10^{38}
双精度型	double	8	-1.7×10^{-308}～1.7×10^{308}
长双精度型	long double	8	-1.7×10^{-308}～1.7×10^{308}



3.7　变量的存储类别

在程序中经常会使用到变量，在 C 程序中可以选择变量的不同存储形式，其存储类别分为静态存储和动态存储。可以通过存储类修饰符来告诉编译器要处理什么样的类型变量，具体主要有自动（auto）、静态（static）、寄存器（register）和外部（extern）4 种。

3.7.1 静态存储与动态存储

根据变量的产生时间，可将其分为静态存储和动态存储。

静态存储是指程序运行时为其分配固定的存储空间，动态存储则是在程序运行期间根据需要动态地分配存储空间。

3.7.2 auto 变量

auto 关键字用于定义一个局部变量为自动的，这意味着每次执行到定义该变量时，都会产生一个新的变量，并且对其重新进行初始化。

【例 3.9】 使用 auto 变量。（**实例位置：资源包\TM\sl\3\9**）

在 AddOne 函数中定义一个 auto 型的整型变量 iInt，在其中对变量进行加 1 操作。之后在主函数 main 中通过显示的提示语句，可以看到调用两次 AddOne 函数的输出，从结果可以看到，在 AddOne 函数中定义整型变量时系统会为其分配内存空间，在函数调用结束时自动释放这些存储空间。

```
#include<stdio.h>

void AddOne()
{
    auto int iInt=1;                    /*定义 auto 整型变量*/
    iInt=iInt+1;                        /*变量加 1*/
    printf("%d\n",iInt);                /*显示结果*/
}
int main()
{
    printf("第一次调用：");              /*显示结果*/
    AddOne();                           /*调用 AddOne 函数*/
    printf("第二次调用：");              /*显示结果*/
    AddOne();                           /*调用 AddOne 函数*/
    return 0;                           /*程序结束*/
}
```

运行程序，显示效果如图 3.16 所示。



图 3.16 使用 auto 变量

注意

事实上，关键字 auto 是可以省略的，如果不特别指定，局部变量的存储方式默认为自动的。

3.7.3 static 变量

static 变量为静态变量，将函数的内部变量和外部变量声明成 static 变量的意义是不一样的（有关函数的内容在本书的后续章节将进行介绍）。不过对于局部变量来说，static 变量是和 auto 变量相对而言的。尽管两者的作用域都仅限于声明变量的函数之中，但是在语句块执行期间，static 变量将始终保持它的值，并且初始化操作只在第一次执行时起作用。在随后的运行过程中，变量将保持语句块上一次执行时的值。

【例 3.10】 使用 static 变量。（**实例位置：资源包\TM\sl\3\10**）

在 AddOne 函数中定义一个 static 型的整型变量 iInt，在其中对变量进行加 1 操作。之后在主函数 main 中通过显示的提示语句，可以看到调用两次 AddOne 函数的输出，从结果中可以发现 static 变量的值保持不变。

```
#include<stdio.h>

void AddOne()
{
	static int iInt=1;                          /*定义 static 整型变量*/
	iInt=iInt+1;                                /*变量加 1*/
	printf("%d\n",iInt);                        /*显示结果*/
}
int main()
{
	printf("第一次调用：");                      /*显示结果*/
	AddOne();                                   /*调用 AddOne 函数*/
	printf("第二次调用：");                      /*显示结果*/
	AddOne();                                   /*调用 AddOne 函数*/
	return 0;                                   /*程序结束*/
}
```

运行程序，显示效果如图 3.17 所示。



图 3.17 使用 static 变量

3.7.4 register 变量

register 变量称为寄存器存储类变量。通过 register 变量，程序员可以把某个局部变量指定存放在计算机的某个硬件寄存器中，而不是内存中。这样做的好处是可以提高程序的运行速度。不过，这只是反映了程序员的主观意愿，实际上，编辑器可以忽略 register 对变量的修饰。

用户无法获得寄存器变量的地址，因为绝大多数计算机的硬件寄存器都不占用内存地址。而且，即使编译器忽略 register，而把变量存放在可设定的内存中，也是无法获取变量的地址的。

如果想有效地利用寄存器 register 关键字，必须像汇编语言程序员那样了解处理器的内部结构，知道可用于存放变量的寄存器的数量、种类以及工作方式。但是，不同计算机对于这些细节可能是不同的，因此，对于一个具备可移植性的程序来说，register 的作用并不大。

下面通过一个实例来介绍寄存器变量的使用方法。

【例 3.11】 使用 register 变量修饰整型变量。（**实例位置：资源包\TM\sl\3\11**）

```
#include<stdio.h>

int main()
{
	register int iInt;			/*定义寄存器整型变量*/
	iInt = 100;
	printf("%d\n",iInt);			/*显示结果*/
	return 0;				/*程序结束*/
}
```

运行程序，显示效果如图 3.18 所示。

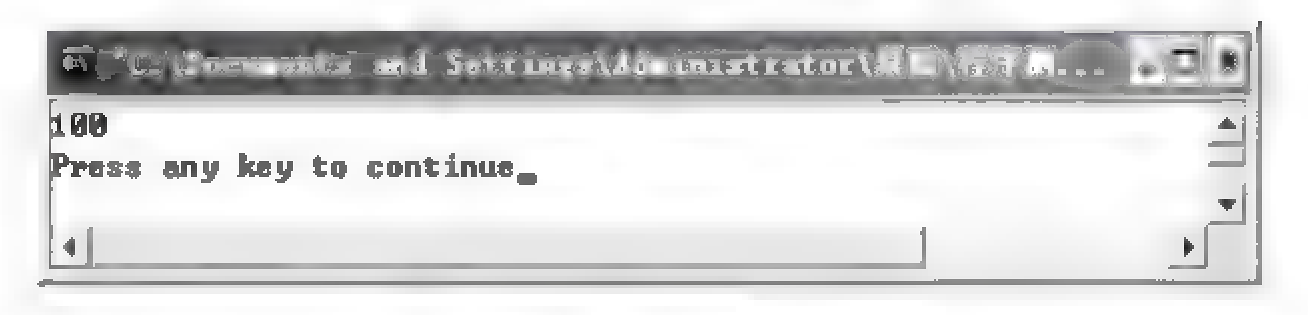


图 3.18 使用 register 变量

3.7.5 extern 变量

extern 变量称为外部存储变量。extern 声明了程序中将要用到但尚未定义的外部变量。通常，外部存储类都用于声明在另一个转换单元中定义的变量。

一个工程是由多个 C 文件组成的。这些源代码文件会分别进行编译，然后链接成一个可执行模块。把这样的一个程序作为一个工程进行管理，并且生成一个工程文件来记录所包含的所有源代码文件。

下面通过一个实例来具体了解一下 extern 变量。

【例 3.12】 使用 extern 变量。（**实例位置：资源包\TM\sl\3\12**）

在本实例中，首先在 Extern1 文件中定义一个外部整型 iExtern 变量，然后在 Extern2 文件中使用 iExtern 变量，并为其进行赋值，将其变量值显示到控制台。

```
/*////////////////////////////////////////////////////////////////////////////////////////////////*/
/*                         在 Extern1 文件中                                */
/*////////////////////////////////////////////////////////////////////////////////////////////////*/
#include<stdio.h>

int main()
{
	extern int iExtern;			/*定义外部整型变量*/
	printf("%d\n",iExtern);			/*显示变量值*/
```

```
    return 0;                               /*程序结束*/
}

/*//////////////////////////////////////////////////////////////////////*/
/*                      在 Extern2 文件中                                 */
/*//////////////////////////////////////////////////////////////////////*/

#include<stdio.h>

int iExtern=100;                            /*定义一个整型变量，为其赋值 100*/
```

运行程序，显示效果如图 3.19 所示。

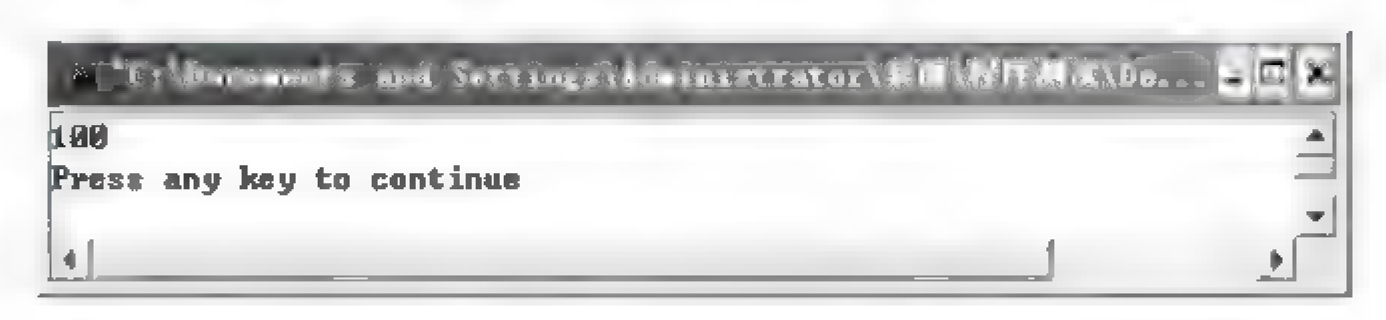


图 3.19　使用 extern 变量

3.8 混合运算



不同类型之间可以进行混合运算，如 10+'a'-1.5+3.2*6。

在进行这样的计算时，不同类型的数据要先转换成同一类型，然后再进行运算，转换的方式如图 3.20 所示。

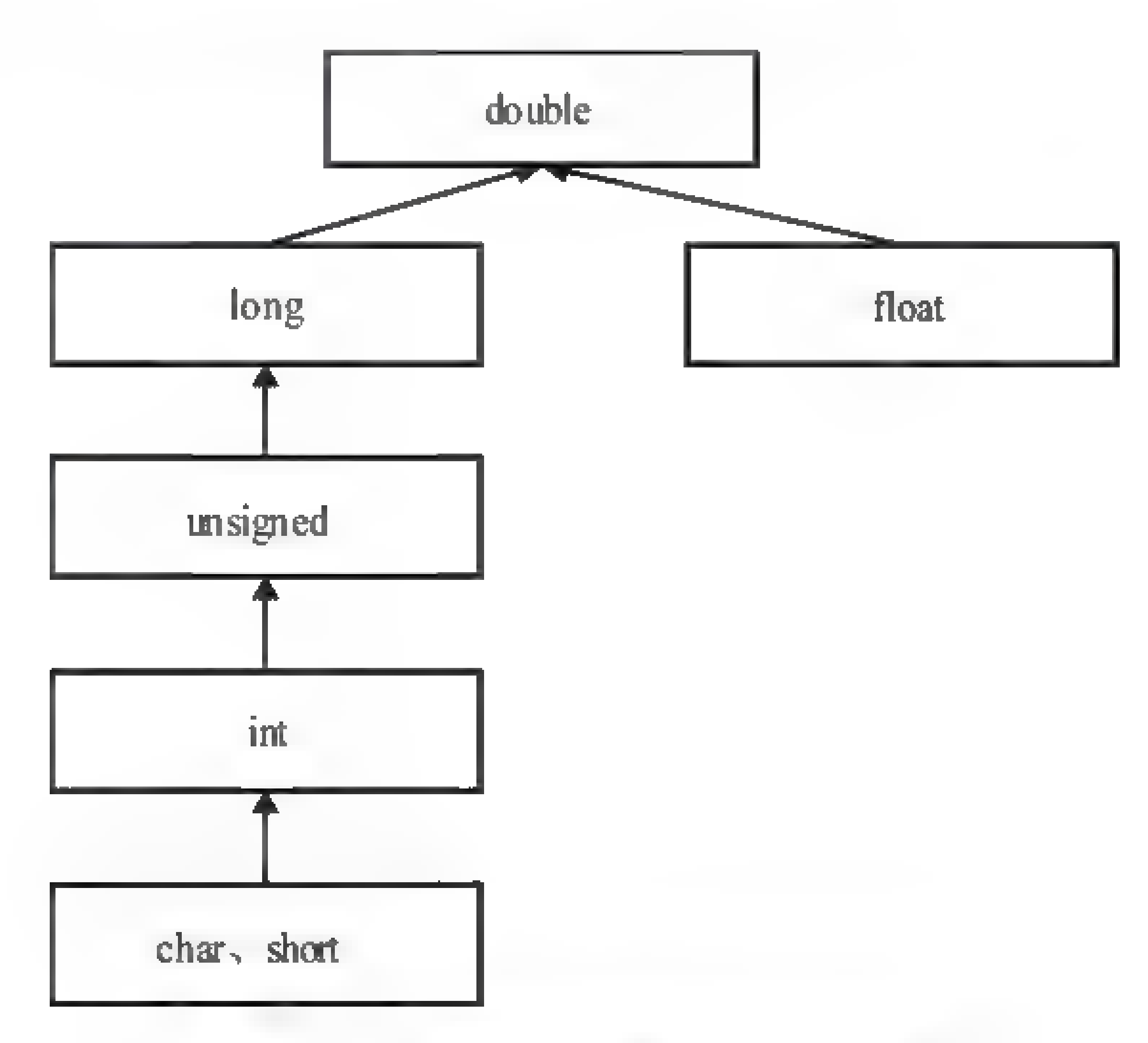


图 3.20　不同类型之间的转换规律

【例 3.13】　混合运算。（**实例位置：资源包\TM\sl\3\13**）

在本实例中，将 int 型变量与 char 型变量、float 型变量进行相加，将其结果存放在 double 类型的 result 变量中，最后使用 printf 函数将其输出。

```
#include<stdio.h>

int main()
{
    int    iInt=1;                      /*定义整型变量*/
    char   cChar='A';                   /*ASCII 码为 65*/
    float fFloat=2.2f;                  /*定义单精度整型变量*/

    double result=iInt+cChar+fFloat;    /*得到相加的结果*/

    printf("%f\n",result);              /*显示变量值*/
    return 0;                           /*程序结束*/
}
```

运行程序，显示效果如图 3.21 所示。



图 3.21　混合运算

3.9　小　　结

本章首先介绍了有关编写程序的一些规范，这些规范虽然不是必需的，但是一个好的编程习惯应该是每一个程序员都必须具备的。

然后介绍了有关常量的内容，引出了有关变量的知识，通过对变量赋值，使得在程序中可以使用变量存储数值。

最后通过介绍变量的存储类别，进一步说明了有关变量的具体使用情况。

3.10　实践与练习

1．定义一个整型变量，为其赋值 345，并使用 printf 输出语句进行输出。（**答案位置：资源包\TM\sl\3\14**）

2．使用字符型变量，在控制台上输出“Fine Day!”。（**答案位置：资源包\TM\sl\3\15**）

3．在自定义的函数中使用 static 静态局部整型变量，计算 3 的立方值。（**答案位置：资源包\TM\sl\3\16**）

4．在文件 1 中定义 extern 外部字符型变量，并为其赋值为'A'。在另一个文件中使用这个变量，将其输出显示到控制台。（**答案位置：资源包\TM\sl\3\17**）

第 4 章

运算符与表达式

（视频讲解：31 分钟）

了解了程序中常用的数据类型后，还应该懂得如何操作这些数据。因此，掌握C语言中各种运算符及其表达式的应用是必不可少的。

本章致力于使读者了解表达式的概念，掌握运算符及相关表达式的使用方法，其中包括赋值运算符、算术运算符、关系运算符、逻辑运算符、位逻辑运算符、逗号运算符和复合赋值运算符，并且通过实例进行相应的练习，加深印象。

通过阅读本章，您可以：

- 了解表达式的使用
- 掌握赋值运算符
- 掌握算术运算符
- 掌握关系运算符
- 掌握逻辑和位逻辑运算符
- 掌握逗号运算符的使用方式



4.1 表 达 式

表达式是C语言的主体。在C语言中，表达式由操作符和操作数组成。最简单的表达式可以只含有一个操作数。根据表达式所含操作符的个数，可以把表达式分为简单表达式和复杂表达式两种，简单表达式是只含有一个操作符的表达式，而复杂表达式是包含两个或两个以上操作符的表达式。

下面通过几个表达式进行观察：

```
5+5
iNumber+9
iBase+(iPay*iDay)
```

表达式本身什么事情也不做，只是返回结果值。在程序不对返回的结果值进行任何操作的情况下，返回的结果值不起任何作用，也就是忽略返回的值。

表达式产生的作用主要有以下两种情况：

☑ 放在赋值语句的右侧（本章要讲解）。

☑ 放在函数的参数中（将在“函数”一章中进行讲解）。

表达式返回的结果值是有类型的。表达式隐含的数据类型取决于组成表达式的变量和常量的类型。



说明

每个表达式的返回值都具有逻辑特性。如果返回值是非零的，那么该表达式返回真值，否则返回假值。通过这个特点，可以将表达式放在用于控制程序流程的语句中，这样就构建了条件表达式。

【例 4.1】 掌握表达式的使用。(**实例位置：资源包\TM\sl\4\1**)

本实例声明了 3 个整型变量，其中有对变量赋值为常数，还有将表达式的结果赋值给变量，最后将变量的值显示在屏幕上。

```
#include<stdio.h>
int main()
{
	int iNumber1,iNumber2,iNumber3;                    /*声明变量*/
	iNumber1=3;                                        /*为变量赋值*/
	iNumber2=7;

	printf("the first number is :%d\n",iNumber1);      /*显示变量值*/
	printf("the second number is :%d\n",iNumber2);

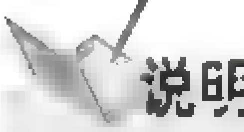
	iNumber3=iNumber1+10;                              /*在表达式中利用 iNumber1 变量加上一个常量*/
	printf("the first number add 10 is :%d\n",iNumber3);  /*显示 iNumber3 的值*/
```

```
    iNumber3=iNumber2+10;                                   /*在表达式中用 iNumber2 变量加上一个常量*/
    printf("the second number add 10 is :%d\n",iNumber3);   /*显示 iNumber3 的值*/

    iNumber3=iNumber1+iNumber2;                             /*在表达式中是两个变量进行计算*/
    printf("the result number of first add second is :%d\n",iNumber3);   /*将计算结果输出*/

    return 0;                                               /*程序结束*/
}
```

（1）在程序中，主函数 main 中的第 1 行代码是声明变量的表达式，可以看到使用逗号通过一个表达式声明 3 个变量。



说明

在 C 语言中，逗号既可以作为分隔符，又可以用在表达式中。

① 逗号作为分隔符使用时，用于间隔说明语句中的变量或函数中的参数。如在上面程序中声明变量时，就属于在语句中使用逗号，将 iNumber1、iNumber2 和 iNumber3 变量进行分隔声明。使用代码举例如下：

```
int iNumber1, iNumber2;                     /*使用逗号分隔变量*/
printf("the number is %d",iResult);         /*使用逗号分隔参数*/
```

② 将逗号用在表达式中，可以将若干个独立的表达式联结在一起。其一般的表现形式如下：

表达式 1,表达式 2,表达式 3…

其运算过程就是先计算表达式 1，然后计算表达式 2……依次计算下去。在循环语句中，逗号就可以在 for 语句中使用，例如：

```
for(i=0,j=100;i<j;i++,j--)                  /*在 for 语句中，使用逗号将表达式进行分隔*/
{
    k=i+j;
}
```

（2）接下来的语句是使用常量为变量赋值的表达式，其中“iNumber1=3;”是将常量 3 赋值给 iNumber1，“iNumber2=7;”语句是将 7 赋值给 iNumber2，然后通过输出语句 printf 显示这两个变量的值。

（3）在语句“iNumber3=iNumber1+10;”中，表达式将变量 iNumber1 与常量 10 相加，然后将返回的值赋给 iNumber3 变量，之后使用输出函数 printf 将 iNumber3 变量的值进行显示。接下来将变量 iNumber2 与常量 10 相加，进行相同的操作。

（4）在语句“iNumber3=iNumber1+iNumber2;”中，可以看到表达式中是两个变量进行相加，同样返回相加的结果，将其值赋给变量 iNumber3，最后输出显示结果。

运行程序，显示效果如图 4.1 所示。

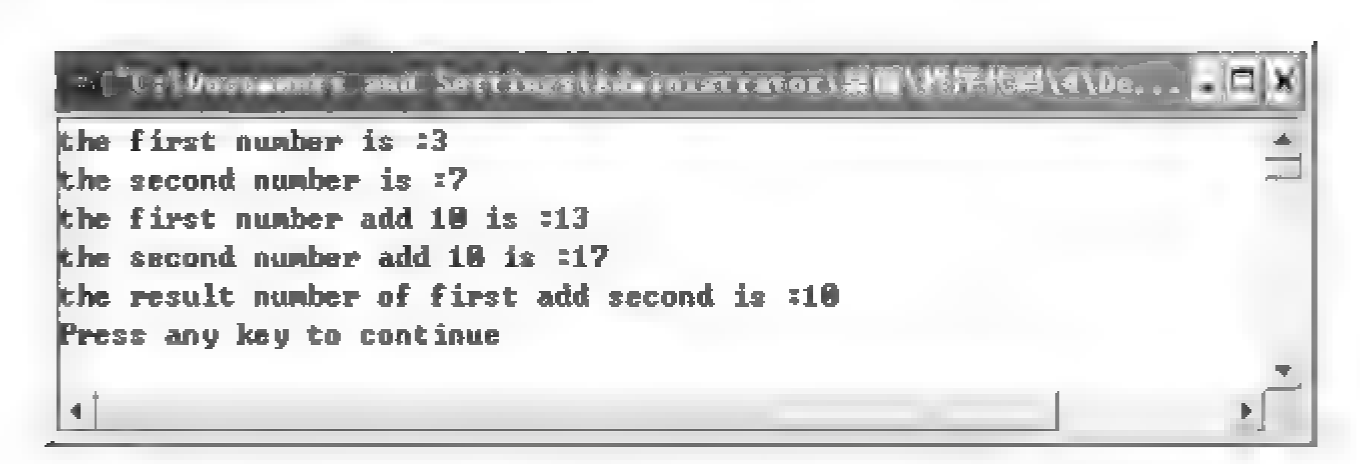


图 4.1　程序输出结果



4.2　赋值运算符与赋值表达式

在程序中常常遇到的赋值符号“=”就是赋值运算符，其作用就是将一个数据赋给一个变量。例如：

```
iAge=20;
```

这就是一次赋值操作，是将常量 20 赋给变量 iAge。同样也可以将一个表达式的值赋给一个变量。例如：

```
Total=Counter*3;
```

下面进行详细的讲解。

4.2.1　变量赋初值

在声明变量时，可以为其赋一个初值，就是将一个常数或者一个表达式的结果赋值给一个变量，变量中保存的内容就是这个常量或者赋值语句中表达式的值。这就是为变量赋初值。

1．将常数赋值给变量

先来看一下将常数赋值给变量的情况。一般形式如下：

```
类型 变量名 = 常数;
```

其中的变量名也称为变量的标识符。通过变量赋初值的一般形式，以下是相关的代码实例：

```
char cChar ='A';
int iFirst=100;
float fPlace=1450.78f;
```

2．通过赋值表达式为变量赋初值

赋值语句把一个表达式的结果值赋给一个变量。一般形式如下：

```
类型 变量名 = 表达式;
```

可以看到，其一般形式与常数赋值的一般形式是相似的，例如：

```
int iAmount= 1+2;
float fPrice= fBase+Day*3;
```

在上面的举例中，得到赋值的变量 iAmount 和 fPrice 称为左值，因为它出现的位置在赋值语句的左侧。产生值的表达式称为右值，因为它出现的位置在表达式的右侧。

注意

这是一个重要的区别，并不是所有的表达式都可以作为左值，如常数只可以作为右值。

在声明变量的同时直接为其赋值的操作称为赋初值，也就是变量的初始化。先将变量声明，再进行变量的赋值操作也是可以的。例如：

```
int iMonth;                                        /*声明变量*/
iMonth= 12;                                        /*为变量赋值*/
```

【例 4.2】 为变量赋初值。（**实例位置：资源包\TM\sl\4\2**）

为变量赋初值的操作是编程时常见的操作。在本实例中，模拟钟点工的计费情况，使用赋值语句和表达式得出钟点工工作 8 个小时后所得的薪水。

```
#include<stdio.h>

int main()
{
    int iHoursWorded=8;                            /*定义变量，为变量赋初值，表示工作时间*/
    int iHourlyRate;                               /*声明变量，表示一个小时的薪水*/
    int iGrossPay;                                 /*声明变量，表示得到的工资*/

    iHourlyRate=13;                                /*为变量赋值*/
    iGrossPay=iHoursWorded*iHourlyRate;            /*将表达式的结果赋值给变量*/

    printf("The HoursWorded is: %d\n",iHoursWorded);    /*显示工作时间变量*/
    printf("The HourlyRate is: %d\n",iHourlyRate);      /*显示一个小时的薪水*/
    printf("The GrossPay is: %d\n",iGrossPay);          /*显示工作所得的工资*/

    return 0;                                      /*程序结束*/
}
```

（1）钟点工的薪水是一个小时的工薪×工作的小时数量，因此在程序中需要 3 个变量来表示这个钟点工薪水的计算过程。iHoursWorded 表示工作的时间，一般的工作时间都是固定的，在这里为其赋初值为 8，表示 8 个小时。iHourlyRate 表示一个小时的工薪。iGrossPay 表示钟点工工作 8 个小时后，应该得到的工资。

（2）工资是可以变化的，iHourlyRate 变量声明之后，为其设定工资，设定为一个小时为 13。根据步骤（1）中计算钟点工薪水的公式，得到总工薪的表达式，将表达式的结果保存在 iGrossPay 变量中。

（3）最后通过输出函数，将变量的值和计算的结果在屏幕上显示。

运行程序，显示效果如图 4.2 所示。



图 4.2　为变量赋初值

4.2.2　自动类型转换

数值类型有很多种，如字符型、整型、长整型和实型等，因为这些类型的变量、长度和符号特性都不同，所以取值范围也不同。混合使用这些类型时会出现什么情况呢？第 3 章已经对此有所介绍。

在 C 语言中默认存在一些自动类型转换规则。根据这些转换规则，数值类型变量可以混合使用。如果把比较短的数值类型变量的值赋给比较长的数值类型变量，那么比较短的数值类型变量中的值会升级为比较长的数值类型，数据信息不会丢失。但是，如果把较长的数值类型变量的值赋给比较短的数值类型变量，那么数据就会降低级别显示，当数据大小超过比较短的数值类型的可表示范围时，就会发生数据截断。

有些编译器遇到这种情况时就会发出警告信息，例如：

```
float i=10.1f;
int j=i;
```

此时编译器会发出警告，如图 4.3 所示。

```
warning C4244: 'initializing' : conversion from 'float ' to 'int ', possible loss of data
```

图 4.3　程序警告

4.2.3　强制类型转换

通过自动类型转换的介绍得知，如果数据类型不同，系统会根据不同情况自动进行类型转换，但此时编译器会提示警告信息。这时如果使用强制类型转换告知编译器，就不会出现警告。

强制类型转换的一般形式如下：

```
(类型名) (表达式)
```

例如，在上述不同变量类型转换时使用强制类型转换：

```
float i=10.1f;
int j= (int)i;                                  /*进行强制类型转换*/
```

在代码中可以看到，在变量前使用包含要转换类型的括号，就对变量进行了强制类型转换。

【例 4.3】　显示类型转换的结果。（**实例位置：资源包\TM\sl\4\3**）

在本实例中，通过不同类型变量之间的赋值，将赋值操作后的结果进行输出，观察类型转换后的结果。

```
#include<stdio.h>

int main()
{
    char cChar;                              /*字符型变量*/
    short int iShort;                        /*短整型变量*/
    int iInt;                                /*整型变量*/
    float fFloat=70000;                      /*单精度浮点型*/

    cChar=(char)fFloat;                      /*强制转换赋值*/
    iShort=(short)fFloat;
    iInt=(int)fFloat;

    printf("the char is: %c\n",cChar);       /*输出字符变量值*/
    printf("the short is: %ld\n",iShort);    /*输出短整型变量值*/
    printf("the int is: %d\n",iInt);         /*输出整型变量值*/
    printf("the float is: %f\n",fFloat);     /*输出单精度浮点型变量值*/

    return 0;                                /*程序结束*/
}
```

本实例定义了一个单精度浮点型变量，然后通过强制转换将其赋给不同类型的变量。因为是由高的级别向低的级别转换，所以可能会出现数据的丢失，在使用强制转换时要注意此问题。

运行程序，显示效果如图 4.4 所示。

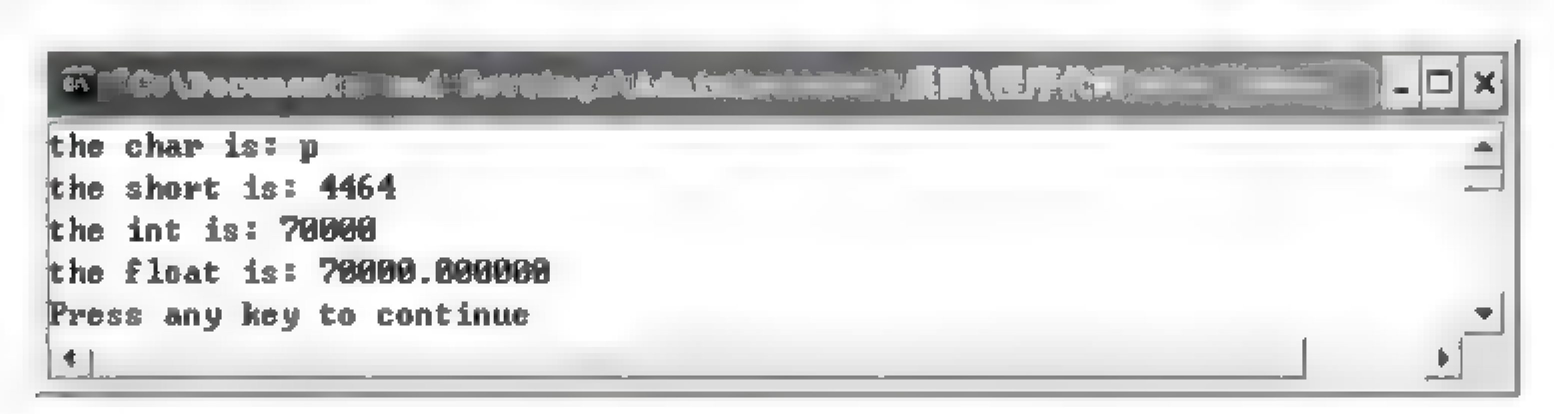


图 4.4　显示类型转换的结果

4.3　算术运算符与算术表达式



C 语言中有两个单目算术运算符和 5 个双目算术运算符。在双目运算符中，乘法、除法和取模运算符比加法和减法运算符的优先级高。单目正和单目负运算符的优先级最高。下面详细进行介绍。

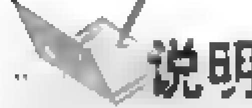
4.3.1　算术运算符

算术运算符包括两个单目运算符（正和负）和 5 个双目运算符（即乘法、除法、取模、加法和减法）。具体符号和对应的功能如表 4.1 所示。

表 4.1　算术运算符

符　　号	功　　能	符　　号	功　　能
+	单目正	%	取模
-	单目负	+	加法
*	乘法	-	减法
/	除法		

在上述算术运算符中，取模运算符“%”用于计算两个整数相除得到的余数，并且取模运算符的两侧均为整数，如 7%4 的结果是 3。


说明

其中的单目正运算符是冗余的，也就是为了与单目负运算符构成一对而存在的。单目正运算符不会改变任何数值，如不会将一个负值表达式改为正。

注意

运算符“-”作为减法运算符，此时为双目运算符，如 5-3。“-”也可作负值运算符，此时为单目运算，如-5 等。

4.3.2　算术表达式

在表达式中使用算术运算符，则该表达式称为算术表达式。下面是一些算术表达式的例子，其中使用的运算符就是表 4.1 中所列出的算术运算符：

```
Number=(3+5)/Rate;
Height= Top-Bottom+1;
Area=Height * Width;
```

需要说明的是，两个整数相除的结果为整数，如 7/4 的结果为 1，舍去的是小数部分。但是，如果其中的一个数是负数时会出现什么情况呢？此时机器会采取“向零取整”的方法，即为-1，取整后向 0 靠拢。

注意

如果用+、-、*、/ 运算的两个数中有一个为实数，那么结果是 double 型，这是因为所有实数都按 double 型进行运算。

【例 4.4】　使用算术表达式计算摄氏温度。（**实例位置：资源包\TM\sl\4\4**）

在本实例中，通过在表达式中使用上面介绍的算术运算符，完成摄氏温度计算，即把用户的华氏温度换算为摄氏温度，然后显示出来。

```
#include<stdio.h>
int main()
```

```
{
    int iCelsius,iFahrenheit;                  /*声明两个变量*/
    printf("Please enter temperature :\n");    /*显示提示信息*/
    scanf("%d",&iFahrenheit);                  /*在键盘上输入华氏温度*/
    iCelsius=5*(iFahrenheit-32)/9;             /*通过算术表达式进行计算，并将结果赋值*/

    printf("Temperature is :");                /*显示提示信息*/
    printf("%d",iCelsius);                     /*显示摄氏温度*/
    printf(" degrees Celsius\n");              /*显示提示信息*/
    return 0;                                  /*程序结束*/
}
```

（1）在主函数 main 中声明两个整型变量，iCelsius 表示摄氏温度，iFahrenheit 表示华氏温度。

（2）使用 printf 函数显示提示信息。之后使用 scanf 函数获得在键盘上输入的数据，其中%d 是格式字符，用来表示输入有符号的十进制整数，这里输入 80。

（3）利用算术表达式，将获得的华氏温度转换成摄氏温度。最后将转换的结果进行输出，可以看到 80 是用户输入的华氏温度，而 26 是计算后输出的摄氏温度。

运行程序，显示效果如图 4.5 所示。

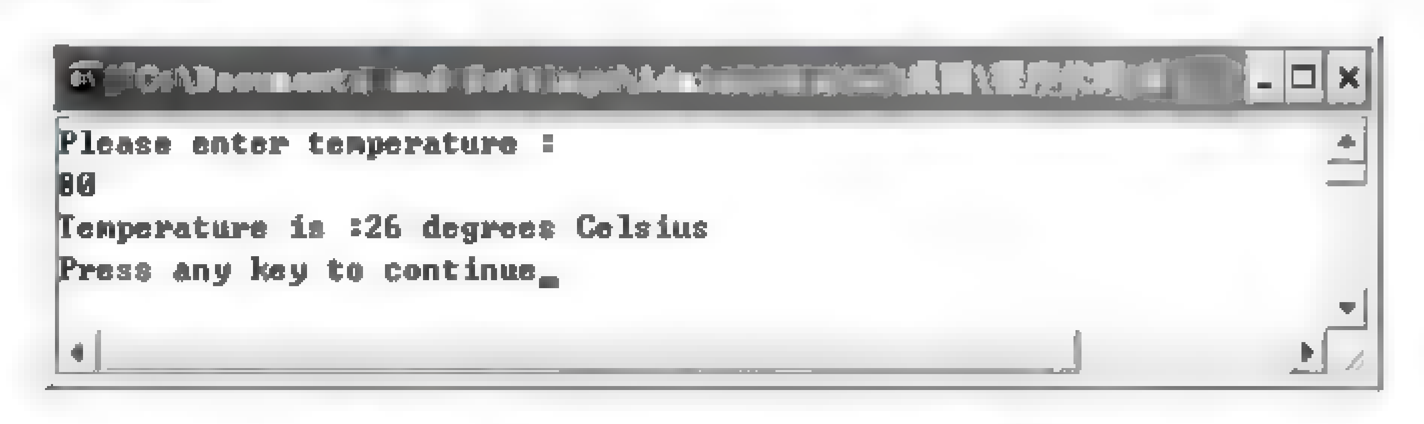


图 4.5 使用算术表达式计算摄氏温度

4.3.3 优先级与结合性

C 语言中规定了各种运算符的优先级和结合性，首先来看一下有关算术运算符的优先级。

1．算术运算符的优先级

在进行表达式求值时，通常会按照运算符的优先级别从高到低依次执行。在算术运算符中，*、/、%的优先级别高于+、-的级别。例如，如果在表达式中同时出现*和+，那么应先运算乘法：

```
R=x+y*z;
```

在上述表达式中，因为*比+的优先级高，所以会先进行 y*z 的运算，最后加上 x。

说明

在表达式中常会出现这样的情况，例如，要进行 a+b，再将结果与 c 相乘，一不小心将表达式写为 a+b*c。因为*的优先级高于+，这样的话就会先执行乘法运算，显然不是期望得到的结果。这时应该怎么办呢？可以使用括号“( )”将+运算级别提高，使其先进行运算，就可以得到预期的结果了，例如解决上式的方法是(a+b)*c。括号可以使其中的表达式先进行运算的原因在于——括号在所有运算符中的优先级别是最高的。

2．算术运算符的结合性

当算术运算符的优先级相同时，结合方向为“自左向右”。例如：

```
a-b+c
```

因为减法和加法的优先级是相同的，所以 b 先与减号相结合，执行 a b 的操作，然后执行加 c 的操作，这样的操作过程就称为“自左向右的结合性”。在后面的介绍中还可以看到“自右向左的结合性”。本章小结处将会给出有关运算符的优先级和结合性的表格，读者可以进行参照。

【例 4.5】 算术运算符的优先级和结合性。（**实例位置：资源包\TM\sl\4\5**）

在本实例中，通过不同运算符的优先级和结合性，使用 printf 函数显示最终的计算结果，根据结果体会优先级和结合性的概念。

```
#include<stdio.h>

int main()
{
	int iNumber1,iNumber2,iNumber3,iResult=0;	/*声明整型变量*/
	iNumber1=20;					/*为变量赋值*/
	iNumber2=5;
	iNumber3=2;

	iResult=iNumber1+iNumber2-iNumber3;		/*加法，减法表达式*/
	printf("the result is : %d\n",iResult);		/*显示结果*/

	iResult=iNumber1-iNumber2+iNumber3;		/*减法，加法表达式*/
	printf("the result is : %d\n",iResult);		/*显示结果*/

	iResult=iNumber1+iNumber2*iNumber3;		/*加法，乘法表达式*/
	printf("the result is : %d\n",iResult);		/*显示结果*/

	iResult=iNumber1/iNumber2*iNumber3;		/*除法，乘法表达式*/
	printf("the result is : %d\n",iResult);		/*显示结果*/

	iResult=(iNumber1+iNumber2)*iNumber3;		/*括号，加法，乘法表达式*/
	printf("the result is : %d\n",iResult);		/*显示结果*/

	return 0;
}
```

（1）在程序中先声明要用到的变量，其中 iResult 的作用是存储计算结果，为其他变量进行赋值。

（2）使用算术运算符完成不同的操作，根据这些不同操作输出的结果来观察优先级与结合性。

- ☑ 根据代码“iResult=iNumber1+iNumber2-iNumber3;”与“iResult=iNumber1-iNumber2+iNumber3;”的结果，可以看出，相同优先级别的运算符根据结合性由左向右进行运算。
- ☑ 语句“iResult=iNumber1+iNumber2*iNumber3;”与上面的语句进行比较，可以看出，不同级别的运算符按照优先级高低依次进行运算。
- ☑ 语句“iResult=iNumber1/iNumber2*iNumber3;”又体现出同优先级的运算符按照结合性进行运算。

☑ 语句“iResult=(iNumber1+iNumber2)*iNumber3;”中使用括号提高优先级，使括号中的表达式先进行运算。从中可知，括号在运算符中具有最高优先级。

运行程序，显示效果如图 4.6 所示。

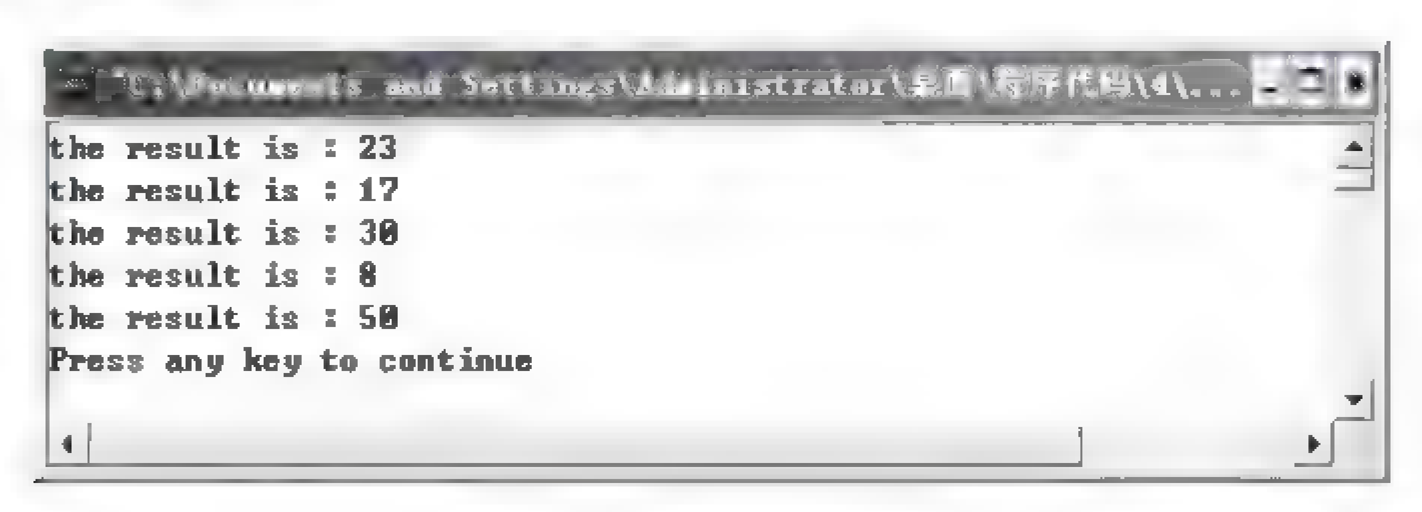


图 4.6　优先级和结合性

4.3.4　自增/自减运算符

在 C 语言中还有两个特殊的运算符，即自增运算符“++”和自减运算符“--”。自增运算符和自减运算符对变量的操作分别是增加 1 和减少 1。

自增运算符和自减运算符可以放在变量的前面或者后面，放在变量前面称为前缀，放在后面称为后缀。使用的一般方法如下：

```
--Counter;                    /*自减前缀符号*/
Grade--;                      /*自减后缀符号*/
++Age;                        /*自增前缀符号*/
Height++;                     /*自增后缀符号*/
```

在上面这些例子中，运算符的前后位置不重要，因为所得到的结果是一样的，自减就是减 1，自增就是加 1。

注意

在表达式内部，作为运算的一部分，两者的用法可能有所不同。如果运算符放在变量前面，那么变量在参加表达式运算之前完成自增或者自减运算；如果运算符放在变量后面，那么变量的自增或者自减运算在变量参加了表达式运算之后完成。

【例 4.6】　比较自增、自减运算符前缀与后缀的不同。（**实例位置：资源包\TM\sl\4\6**）

在本实例中定义一些变量，为变量赋相同的值，然后通过前缀和后缀的操作来观察在表达式中前缀和后缀的不同结果。

```
#include<stdio.h>

int main()
{
    int iNumber1=3;                    /*定义变量，赋值为 3*/
    int iNumber2=3;

    int iResultPreA,iResultLastA;      /*声明变量，得到自增运算的结果*/
```

```
    int iResultPreD,iResultLastD;                    /*声明变量，得到自减运算的结果*/

    iResultPreA=++iNumber1;                          /*前缀自增运算*/
    iResultLastA=iNumber2++;                         /*后缀自增运算*/

    printf("The Addself ...\n");
    printf("the iNumber1 is :%d\n",iNumber1);        /*显示自增运算后自身的数值*/
    printf("the iResultPreA is :%d\n",iResultPreA);  /*得到自增表达式中的结果*/
    printf("the iNumber2 is :%d\n",iNumber2);        /*显示自增运算后自身的数值*/
    printf("the iResultLastA is :%d\n",iResultLastA); /*得到自增表达式中的结果*/

    iNumber1=3;                                      /*恢复变量的值为 3*/
    iNumber2=3;

    iResultPreD=--iNumber1;                          /*前缀自减运算*/
    iResultLastD=iNumber2--;                         /*后缀自减运算*/

    printf("The Deleteself ...\n");
    printf("the iNumber1 is :%d\n",iNumber1);        /*显示自减运算后自身的数值*/
    printf("the iResultPreD is :%d\n",iResultPreD);  /*得到自减表达式中的结果*/
    printf("the iNumber2 is :%d\n",iNumber2);        /*显示自减运算后自身的数值*/
    printf("the iResultLastD is :%d\n",iResultLastD) ; /*得到自减表达式中的结果*/

    return 0;                                        /*程序结束*/
}
```

（1）在程序代码中，定义 iNumber1 和 iNumber2 两个变量用来进行自增、自减运算。

（2）进行自增运算，分为前缀自增和后缀自增。通过程序最终的显示结果可以看到，自增变量 iNumber1 和 iNumber2 的结果同为 4，但是得到表达式结果的两个变量 iResultPreA 和 iResultLastA 却不一样。iResultPreA 的值为 4，iResultLastA 的值为 3，因为前缀自增使得 iResultPreA 变量先进行自增操作，然后进行赋值操作；后缀自增操作是先进行赋值操作，然后进行自增操作。因此两个变量得到表达式的结果值是不一样的。

（3）在自减运算中，前缀自减和后缀自减与自增运算方式是相同的，前缀自减是先进行减 1 操作，然后赋值操作；而后缀自减是先进行赋值操作，再进行自减操作。

运行程序，显示效果如图 4.7 所示。

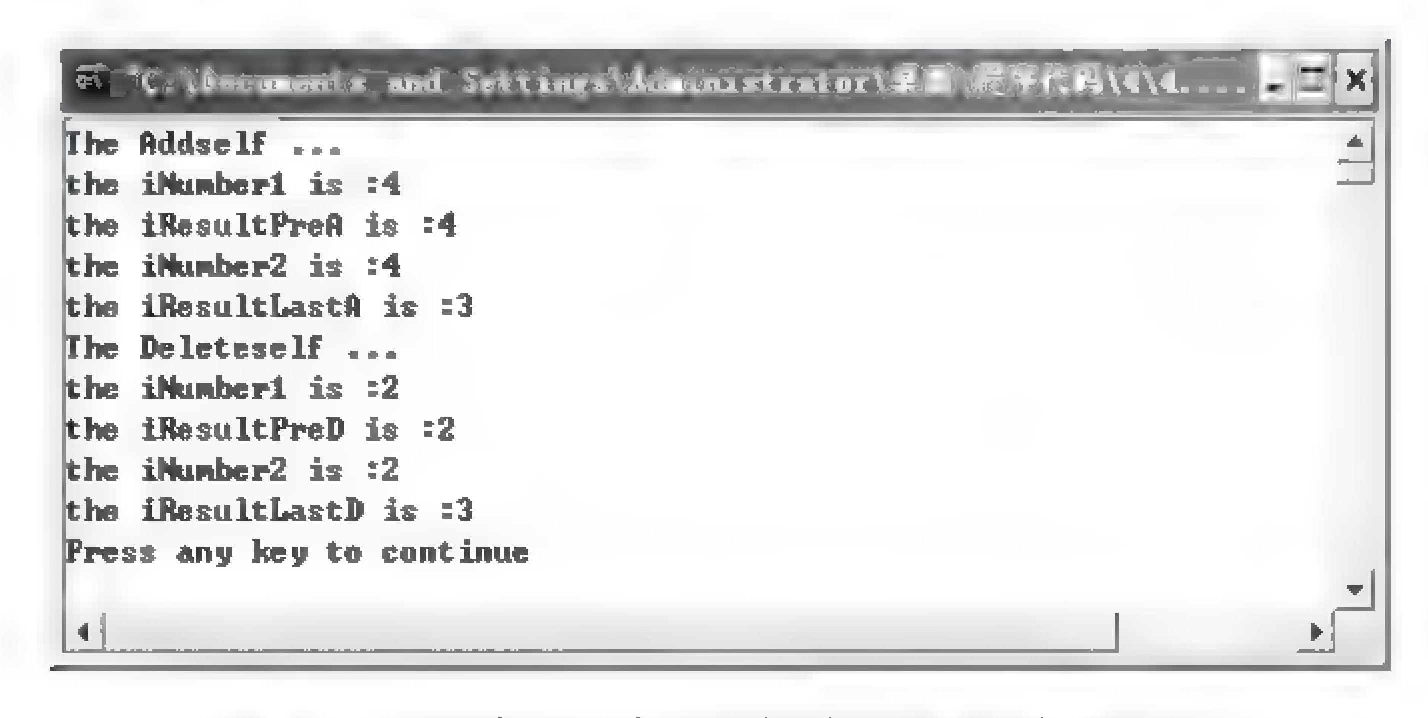


图 4.7　比较自增、自减运算符前缀与后缀的不同

4.4　关系运算符与关系表达式



在数学中，经常需要比较两个数的大小。在 C 语言中，关系运算符的作用就是判断两个操作数的大小关系。

4.4.1　关系运算符

关系运算符包括大于、大于等于、小于、小于等于、等于和不等于，如表 4.2 所示。

表 4.2　关系运算符

符　号	功　能	符　号	功　能
>	大于	<=	小于等于
>=	大于等于	==	等于
<	小于	!=	不等于

注意

符号“>=”（大于等于）与“<=”（小于等于）的意思分别是大于或等于、小于或等于。

4.4.2　关系表达式

关系运算符用于对两个表达式的值进行比较，返回一个真值或者假值。返回真值还是假值，取决于表达式中的值和所用的运算符。其中真值为 1，假值为 0，真值表示指定的关系成立，假值则表示指定的关系不成立。例如：

```
7>5                    /*因为 7 大于 5，所以该关系成立，表达式的结果为真值*/
7>=5                   /*因为 7 大于 5，所以该关系成立，表达式的结果为真值*/
7<5                    /*因为 7 大于 5，所以该关系不成立，表达式的结果为假值*/
7<=5                   /*因为 7 大于 5，所以该关系不成立，表达式的结果为假值*/
7==5                   /*因为 7 不等于 5，所以该关系不成立，表达式的结果为假值*/
7!=5                   /*因为 7 不等于 5，所以该关系成立，表达式的结果为真值*/
```

关系运算符通常用来构造条件表达式，多用在程序流程控制语句中。例如，if 语句是用于判断条件而执行语句块，在其中使用关系表达式作为判断条件，如果关系表达式返回的是真值，则执行下面的语句块；如果为假值，就不去执行。代码如下：

```
if(Count<10)
{
    …                  /*判断条件为真值，执行代码*/
}
```

其中，if(iCount<10)就是判断 iCount 小于 10 这个关系是否成立，如果成立则为真，如果不成立则为假。

注意

在进行判断时，一定要注意等号运算符“==”的使用，千万不要与赋值运算符“=”弄混。如在 if 语句中进行判断，使用的是“=”：

```
if(Amount=100)
{
    …
}
```

上面的代码看上去是在检验变量 Amount 是否等于常量 100，但是事实上没有起到这个效果。因为表达式使用的是赋值运算符“=”而不是等于运算符“==”。赋值表达式 Amount=100，本身也是表达式，其返回值是 100。既然是 100，说明是非零值也就是真值，则该表达式的值始终为真值，没有起到进行判断的作用。如果赋值表达式右侧不是常量 100，而是变量，则赋值表达式的真值或假值就由这个变量的值决定。

因为这两个运算符在语言上存在差别，使用其构造条件表达式时很容易出现错误，新手在编写程序时一定要加以注意。

4.4.3 优先级与结合性

关系运算符的结合性都是自左向右的。使用关系运算符时常常会判断两个表达式的关系，但是由于运算符存在着优先级的问题，因此如果不小心处理则会出现错误。例如，先对一个变量进行赋值，然后判断这个赋值的变量是否不等于一个常数，代码如下：

```
if(Number=NewNum!=10)
{
    …
}
```

因为“!=”运算符比“=”的优先级要高，所以 NewNum!=10 的判断操作会在赋值之前实现，变量 Number 得到的就是关系表达式的真值或者假值，这样并不会按照之前的意愿执行。

前文曾经介绍过括号运算符，其优先级具有最高性，因此应该使用括号来表示需要优先进行计算的表达式，例如：

```
if((Number=NewNum)!=10)
{
    …
}
```

这种写法比较清楚，不会产生混淆，没有人会对代码的含义产生误解。由于这种写法格式比较精确、简洁，因此被多数程序员所接受。

【例 4.7】　关系运算符的使用。(**实例位置：资源包\TM\sl\4\7**)

在本实例中，定义两个变量表示两个学科的分数，使用 if 语句判断两个学科的分数大小，通过 printf 输出函数显示信息，得到比较的结果。

```
#include<stdio.h>

int main()
{
	int iChinese,iEnglish;                      /*定义两个变量，用来保存分数*/
	printf("Enter Chinese score:");             /*提示信息*/
	scanf("%d",&iChinese);                      /*输入分数*/
	printf("Enter English score:");             /*提示信息*/
	scanf("%d",&iEnglish);                      /*输入分数*/

	if(iChinese>iEnglish)                       /*使用关系表达式进行判断*/
	{
		printf("Chinese is better than English\n");
	}
	if(iChinese<iEnglish)                       /*使用关系表达式进行判断*/
	{
		printf("English is better than Chinese\n");
	}
	if(iChinese==iEnglish)                      /*使用关系表达式进行判断*/
	{
		printf("Chinese equal English\n");
	}
	return 0;
}
```

为了可以在键盘上得到两个学科的分数，定义变量 iChinese 和 iEnglish。然后利用 if 语句进行判断，在判断条件中使用了关系表达式，判断分数是否使得表达式成立。如果成立，则返回真值；如果不成立，则返回假值。最后根据真值和假值选择输出语句。

运行程序，显示效果如图 4.8 所示。



图 4.8　关系运算符的使用

4.5　逻辑运算符与逻辑表达式



逻辑运算符根据表达式的真或者假属性返回真值或假值。在 C 语言中，表达式的值非零，那么其值为真。非零的值用于逻辑运算，则等价于 1；假值总是为 0。

4.5.1 逻辑运算符

逻辑运算符有 3 种，如表 4.3 所示。

表 4.3 逻辑运算符

符　号	功　能
&&	逻辑与
||	逻辑或
!	单目逻辑非

注意

表 4.3 中的逻辑与运算符“&&”和逻辑或运算符“||”都是双目运算符。

4.5.2 逻辑表达式

前文介绍过，关系运算符可用于对两个操作数进行比较，使用逻辑运算符可以将多个关系表达式的结果合并在一起进行判断。其一般形式如下：

```
表达式  逻辑运算符  表达式
```

例如，使用逻辑运算符：

```
Result= Func1&&Func2;                 /*Func1 和 Func2 都为真时，结果为真*/
Result= Func1||Func2;                 /*Func1、Func2 其中一个为真时，结果为真*/
Result= !Func2;                       /*如果 Func2 为真，则 Result 为假*/
```

前面已经介绍过，但这里还要做重点强调，不要把逻辑与运算符“&&”和逻辑或运算符“||”与下面要讲的位与运算符“&”和位或运算符“|”混淆。

逻辑与运算符和逻辑或运算符可以用于相当复杂的表达式中。一般来说，这些运算符用来构造条件表达式，用在控制程序的流程语句中，例如在后面章节中要介绍的 if、for、while 语句等。

在程序中，通常使用单目逻辑非运算符“!”把一个变量的数值转换为相应的逻辑真值或者假值，也就是 1 或 0。例如：

```
Result= !!Value;                      /*转换成逻辑值*/
```

4.5.3 优先级与结合性

“&&”和“||”是双目运算符，它们要求有两个操作数，结合方向自左至右；“!”是单目运算符，要求有一个操作数，结合方向自左向右。

逻辑运算符的优先级从高到低依次为单目逻辑非运算符“!”、逻辑与运算符“&&”和逻辑或运算符“||”。

【例 4.8】　逻辑运算符的应用。（**实例位置：资源包\TM\sl\4\8**）

在本实例中，使用逻辑运算符构造表达式，通过输出函数显示表达式的结果，根据结果分析表达式中逻辑运算符的计算过程。

```
#include<stdio.h>

int main()
{
    int iNumber1,iNumber2;                                          /*声明变量*/
    iNumber1=10;                                                    /*为变量赋值*/
    iNumber2=0;

    printf("the 1 is Ture , 0 is False\n");                         /*显示提示信息*/
    printf("5<   iNumber1&&iNumber2 is %d\n",5<iNumber1&&iNumber2); /*显示逻辑与表达式的结果*/
    printf("5<   iNumber1||iNumber2 is %d\n",5<iNumber1||iNumber2); /*显示逻辑或表达式的结果*/
    iNumber2=!!iNumber1;                                            /*得到 iNumber1 的逻辑值*/
    printf("iNumber2 is %d\n",iNumber2);                            /*输出逻辑值*/
    return 0;
}
```

（1）在程序中，先声明两个变量用来进行下面的计算。为变量赋值，iNumber1 的值为 10，iNumber2 的值为 0。

（2）先进行输出信息，说明显示为 1 表示真值，0 表示假值。在 printf 函数中，进行表达式的运算，最后将结果输出。分析表达式 5<iNumber1&&iNumber2，由于“&&”运算符的优先级低于“<”运算符，因此先执行关系判断，之后进行与运算。iNumber1 的值为 10，iNumber2 的值为 0，这个表达式的含义是先计算数值 5 小于 iNumber1 的结果，然后将其结果与 iNumber2 执行逻辑与运算，计算结果为 0。表达式 5<iNumber1||iNumber2 的含义是先计算数值 5 小于 iNumber1 的结果，然后将其结果与 iNumber2 执行逻辑或运算，计算结果为 1。

（3）将 iNumber1 进行两次单目逻辑非运算，得到的是逻辑值，因为 iNumber1 的数值是 10，所以逻辑值为 1。

运行程序，显示效果如图 4.9 所示。

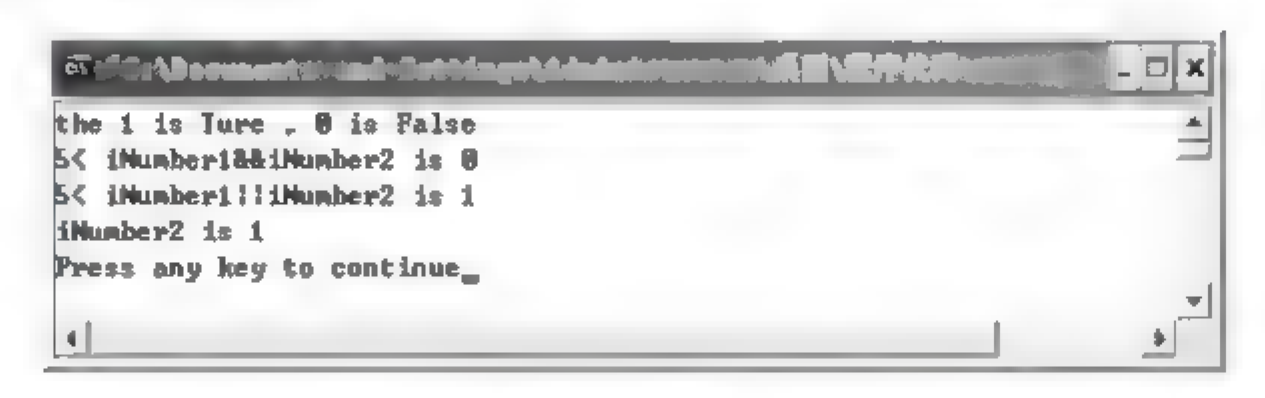


图 4.9　逻辑运算符的应用

4.6　位逻辑运算符与位逻辑表达式



位运算是 C 语言中比较有特色的内容。位逻辑运算符可实现位的设置、清零、取反和取补操作。利用位运算可以实现只有部分汇编语言才能实现的功能。

4.6.1 位逻辑运算符

位逻辑运算符包括位逻辑与、位逻辑或、位逻辑非和取补，如表 4.4 所示。

表 4.4 位逻辑运算符

符 号	功 能
&	位逻辑与
|	位逻辑或
^	位逻辑非
~	取补

表 4.4 中除了最后一个运算符是单目运算符外，其他都是双目运算符，这些运算符只能用于整型表达式。位逻辑运算符通常用于对整型变量进行位的设置、清零和取反，以及对某些选定的位进行检测。

4.6.2 位逻辑表达式

在程序中，位逻辑运算符一般被程序员用作开关标志。较低层次的硬件设备驱动程序，经常需要对输入/输出设备进行位操作。

如下为位逻辑与运算符的典型应用，对某个语句的位设置进行检查：

```
if(Field & BITMASK)
```

语句的含义是：if 语句对后面括号中的表达式进行检测。如果表达式返回的是真值，则执行下面的语句块，否则跳过该语句块，不执行。其中，运算符用来对 BITMASK 变量的位进行检测，判断其是否与 Field 变量的位有相吻合之处。



4.7 逗号运算符与逗号表达式

在 C 语言中，可以用逗号将多个表达式分隔开来。其中，用逗号分隔的表达式被分别计算，并且整个表达式的值是最后一个表达式的值。

逗号表达式称为顺序求值运算符。逗号表达式的一般形式如下：

```
表达式 1,表达式 2,…,表达式 n
```

逗号表达式的求解过程是：先求解表达式 1，再求解表达式 2，一直求解到表达式 n。整个逗号表达式的值是表达式 n 的值。

观察下面使用逗号运算符的代码：

```
Value=2+5,1+2,5+7;
```

在上面的语句中，Value 所得到的值为 7，而非 12。整个逗号表达式的值不应该是最后一个表达

式的值吗？为什么不等于 12 呢？答案在于优先级的问题，由于赋值运算符的优先级比逗号运算符的优先级高，因此先执行赋值的运算。如果要先执行逗号运算，则可以使用括号运算符，代码如下：

```
Value=(2+5,1+2,5+7);
```

使用括号之后，Value 的值为 12。

【例 4.9】　用逗号运算符分隔的表达式。(**实例位置：资源包\TM\sl\4\9**)

在本实例中，通过逗号运算符将其他运算符结合在一起形成表达式，再将表达式的最终结果赋值给变量。根据变量的值分析逗号运算符的计算过程。

```
#include<stdio.h>

int main()
{
    int iValue1,iValue2,iValue3,iResult;                /*声明变量，使用逗号运算符*/

    /*为变量赋值*/
    iValue1=10;
    iValue2=43;
    iValue3=26;
    iResult=0;

    iResult=iValue1++,--iValue2,iValue3+4;             /*计算逗号表达式*/
    printf("the result is :%d\n",iResult);             /*将结果输出显示*/

    iResult=(iValue1++,--iValue2,iValue3+4);           /*计算逗号表达式*/
    printf("the result is :%d\n",iResult);             /*将结果输出显示*/
    return 0;                                          /*程序结束*/
}
```

（1）在程序代码的开始处，声明变量时使用了逗号运算符，分隔声明变量。前文已经对此有所介绍。

（2）将前面使用逗号分隔声明的变量进行赋值。在逗号表达式中，赋值的变量进行各自的计算，变量 iResult 得到表达式的结果。这里需要注意的是，通过输出可以看到 iResult 的值为 10，从前面的讲解知道因为逗号表达式没有使用括号运算符，所以 iResult 得到第一个表达式的值。在第一个表达式中，iValue1 变量进行的是后缀自加操作，于是 iResult 先得到 iValue1 的值，iValue1 再进行自加操作。

（3）在第二个表达式中，由于使用了括号运算符，因此 iResult 变量得到的是第 3 个表达式 iValue3+4 的值，iResult 变量赋值为 30。

运行程序，显示效果如图 4.10 所示。



图 4.10　用逗号运算符分隔的表达式



4.8 复合赋值运算符

复合赋值运算符是 C 语言中独有的，实际这种操作是一种缩写形式，可使得变量操作的描述方式更为简洁。例如，在程序中为一个变量赋值：

```
Value=Value+3;
```

这个语句是对一个变量进行赋值操作，值为这个变量本身与一个整数常量 3 相加的结果值。使用复合赋值运算符可以实现同样的操作。例如，上面的语句可以改写成：

```
Value+=3;
```

这种描述更为简洁。关于上面两种实现相同操作的语句，赋值运算符和复合赋值运算符的区别在于后者：

☑ 简化了程序，使程序精练。

☑ 提高了编译效率。

对于简单赋值运算符，如 Func=Func+1 中，表达式 Func 计算两次；对于复合赋值运算符，如 Func+=1 中，表达式 Func 仅计算一次。一般来说，这种区别对于程序的运行没有太大的影响，但是如果表达式中存在某个函数的返回值，那么函数将被调用两次。

【例 4.10】 使用复合赋值运算符简化赋值运算。（**实例位置：资源包\TM\sl\4\10**）

```
#include<stdio.h>

int main()
{
	int iTotal,iValue,iDetail;				/*声明变量*/
	iTotal=100;						/*为变量赋值*/
	iValue=50;
	iDetail=5;

	iValue*=iDetail;					/*计算得到 iValue 变量值*/
	iTotal+=iValue;						/*计算得到 iTotal 变量值*/
	printf("Value is: %d\n",iValue);			/*显示计算结果*/
	printf("Total is: %d\n",iTotal);
	return 0;
}
```

从程序代码中可以看到，语句 iValue*=iDetail 中使用复合赋值运算符，表示的意思是 iValue 的值等于 iValue*iDetail 的结果。而 iTotal+=iValue 表示的是 iTotal 的值等于 iTotal+iValue 的结果。最后将结果显示输出。

运行程序，显示效果如图 4.11 所示。

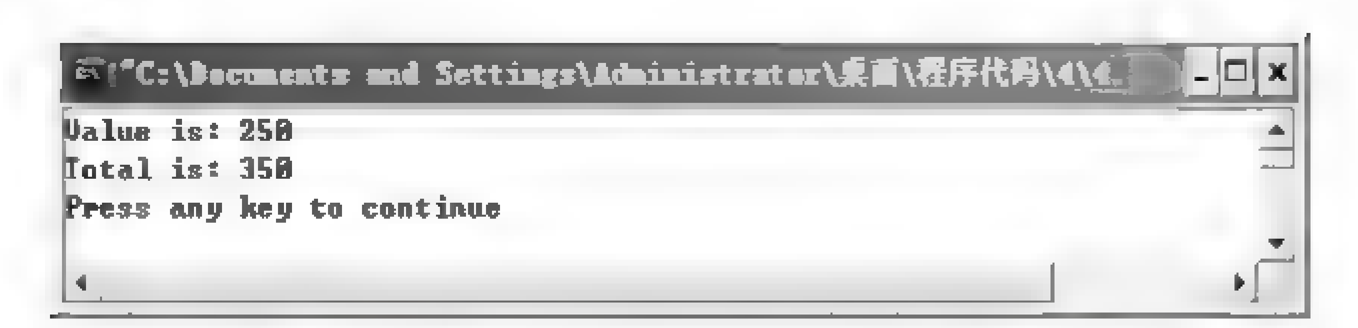


图 4.11　使用复合赋值运算符简化赋值运算

4.9　小　　结

本章介绍了程序的各种运算符与表达式。首先介绍了表达式的概念，帮助读者了解后续章节所需要的准备知识，然后分别介绍了赋值运算符、算术运算符、关系运算符、逻辑运算符、位逻辑运算符和逗号运算符，最后讲解了如何使用复合运算符简化程序的编写。

同时为了方便读者，这里按照优先级从小到大的排列顺序列出了 C 语言中运算符的优先级和结合性，如表 4.5 所示。

表 4.5　运算符的优先级和结合性

<table>
<tr><th>优先级</th><th>运算符</th><th>含义</th><th>结合性</th></tr>
<tr><td rowspan="4">1</td><td>()</td><td>圆括号</td><td>自左向右</td></tr>
<tr><td>[]</td><td>下标运算符</td><td></td></tr>
<tr><td>></td><td>指向结构体成员运算符</td><td></td></tr>
<tr><td>.</td><td>结构体成员运算符</td><td></td></tr>
<tr><td rowspan="8">2</td><td>!</td><td>逻辑非运算符（单目运算符）</td><td>自右向左</td></tr>
<tr><td>~</td><td>按位取反运算符（单目运算符）</td><td></td></tr>
<tr><td>++</td><td>自增运算符（单目运算符）</td><td></td></tr>
<tr><td>--</td><td>自减运算符（单目运算符）</td><td></td></tr>
<tr><td>-</td><td>负号运算符（单目运算符）</td><td></td></tr>
<tr><td>*</td><td>指针运算符（单目运算符）</td><td></td></tr>
<tr><td>&</td><td>地址与运算符（单目运算符）</td><td></td></tr>
<tr><td>sizeof</td><td>长度运算符（单目运算符）</td><td></td></tr>
<tr><td>3</td><td>*、/、%</td><td>乘法、除法、求余运算符</td><td>自左向右</td></tr>
<tr><td>4</td><td>+、-</td><td>加法、减法运算符</td><td></td></tr>
<tr><td>5</td><td><<、>></td><td>左移、右移运算符</td><td></td></tr>
<tr><td>6</td><td><、<=、>、>=</td><td>小于、小于或等于、大于、大于或等于运算符</td><td></td></tr>
<tr><td>7</td><td>==、!=</td><td>等于、不等于运算符</td><td></td></tr>
<tr><td>8</td><td>&</td><td>按位与运算符</td><td></td></tr>
<tr><td>9</td><td>^</td><td>按位异或运算符</td><td></td></tr>
<tr><td>10</td><td>|</td><td>按位或运算符</td><td></td></tr>
<tr><td>11</td><td>&&</td><td>逻辑与运算符</td><td></td></tr>
<tr><td>12</td><td>||</td><td>逻辑或运算符</td><td></td></tr>
</table>

续表

优先级	运算符	含义	结合性
13	?:	条件运算符（三目运算符）	自右向左
14	=、+=、-=、*=、/=、%=、>>=、<<=、&=、^=、|=	赋值运算符	
15	,	逗号运算符（顺序求值运算符）	自左向右

4.10 实践与练习

1. 使用复合运算符计算 a+=a*=a/=a-6。**（答案位置：资源包\TM\sl\4\11）**

2. 定义一个变量，赋值为 6，经过前缀自加、后缀自加、前缀自减和后缀自减，得到每一次运算的结果。**（答案位置：资源包\TM\sl\4\12）**

第 5 章

常用的数据输入/输出函数

（视频讲解：40 分钟）

与其他高级语言一样，C 语言的语句是用来向计算机系统发出操作指令的。当需要程序按照要求执行指令时，先要使用输入语句给程序发送指示。当程序解决了一个问题之后，还要使用输出语句将计算的结果显示出来。

本章致力于使读者了解有关语句的概念，掌握如何对程序的输入/输出进行操作，并且对这些输入和输出操作按照不同的方式进行讲解。

通过阅读本章，您可以：

- 了解有关语句的概念
- 掌握单个字符数据的输入/输出操作
- 掌握字符串的输入/输出操作
- 掌握数据的格式化输入和输出方法



5.1 语 句

C 语言的语句用来向计算机系统发出操作指令。一条语句经过编译后，会产生若干条机器指令。实际程序中通常包含若干条语句，用于完成一定的操作任务。

注意

在编写程序时，声明部分不能算作语句。例如，"int iNumber;"就不是一条语句，因为不产生机器的操作，只是对变量提前进行了定义。

通过前面的介绍可知，程序包括声明部分和执行部分，其中执行部分由语句组成。



5.2 字符数据输入/输出

本节将介绍 C 标准 I/O 函数库中最简单也最容易理解的字符输入/输出函数 getchar 和 putchar。

5.2.1 字符数据输出

输出字符数据使用的是 putchar 函数，作用是向显示设备输出一个字符。其语法格式如下：

```
int putchar(int ch);
```

使用该函数时，要添加头文件 stdio.h。其中，参数 ch 为要进行输出的字符，可以是字符型变量或整型变量，也可以是常量。例如，输出一个字符 A 的代码如下：

```
putchar('A');
```

使用 putchar 函数也可以输出转义字符，如输出字符 A：

```
putchar('\101');
```

【例 5.1】 使用 putchar 函数实现字符数据输出。（**实例位置：资源包\TM\sl\5\1**）

在程序中使用 putchar 函数，输出字符串"Hello"，并在输出完毕之后换行。

```
#include<stdio.h>

int main()
{
    char cChar1,cChar2,cChar3,cChar4;                /*声明变量*/
    cChar1='H';                                      /*为变量赋值*/
    cChar2='e';
    cChar3='l';
    cChar4='o';
```

```
    putchar(cChar1);                                /*输出字符变量*/
    putchar(cChar2);
    putchar(cChar3);
    putchar(cChar3);
    putchar(cChar4);
    putchar('\n');                                  /*输出转义字符*/
    return 0;
}
```

（1）要使用 putchar 函数，首先要包含头文件 stdio.h。

（2）声明字符型变量，用来保存要输出的字符。

（3）为字符变量赋值时，因为 putchar 函数只能输出一个字符，如果要输出字符串，就需要多次调用 putchar 函数。

（4）当字符串输出完毕之后，使用 putchar 函数输出转义字符“\n”进行换行操作。

运行程序，显示效果如图 5.1 所示。



图 5.1　使用 putchar 函数实现字符数据输出

5.2.2　字符数据输入

字符数据输入使用的是 getchar 函数，其作用是从终端（输入设备）输入一个字符。getchar 与 putchar 函数的区别在于 getchar 函数没有参数。

getchar 函数的语法格式如下：

```
int getchar();
```

使用 getchar 函数时也要添加头文件 stdio.h，函数的值就是从输入设备得到的字符。例如，从输入设备得到一个字符赋给字符变量 cChar，代码如下：

```
cChar=getchar();
```

注意

getchar 函数只能接收一个字符，该字符可以赋给一个字符变量或整型变量，也可以不赋给任何变量，只是作为表达式的一部分，如“putchar(getchar());”。这里，getchar 函数作为 putchar 函数的参数，通过 getchar 函数从输入设备得到一个字符，然后通过 putchar 函数将字符输出。

【例 5.2】　使用 getchar 函数实现字符数据输入。（**实例位置：资源包\TM\sl\5\2**）

在本实例中，使用 getchar 函数获取在键盘上输入的字符，再利用 putchar 函数进行输出。本实例演示了将 getchar 函数作为 putchar 函数表达式的一部分，输入和输出字符的方式。

```
#include<stdio.h>

int main()
{
    char cChar1;                                /*声明变量*/
    cChar1=getchar();                           /*在输入设备得到字符*/
    putchar(cChar1);                            /*输出字符*/
    putchar('\n');                              /*输出转义字符换行*/
    getchar();                                  /*得到回车字符*/
    putchar(getchar());                         /*得到输入字符，直接输出*/
    putchar('\n');                              /*换行*/
    return 0;                                   /*程序结束*/
}
```

（1）要使用 getchar 函数，首先要包括头文件 stdio.h。

（2）声明变量 cChar1，通过 getchar 函数得到输入的字符，赋值给 cChar1 字符型变量，然后使用 putchar 函数将变量输出。

（3）使用 getchar 函数得到输入过程中的回车符。

（4）在 putchar 函数的参数位置调用 getchar 函数，得到字符，并将得到的字符输出。

运行程序，显示效果如图 5.2 所示。

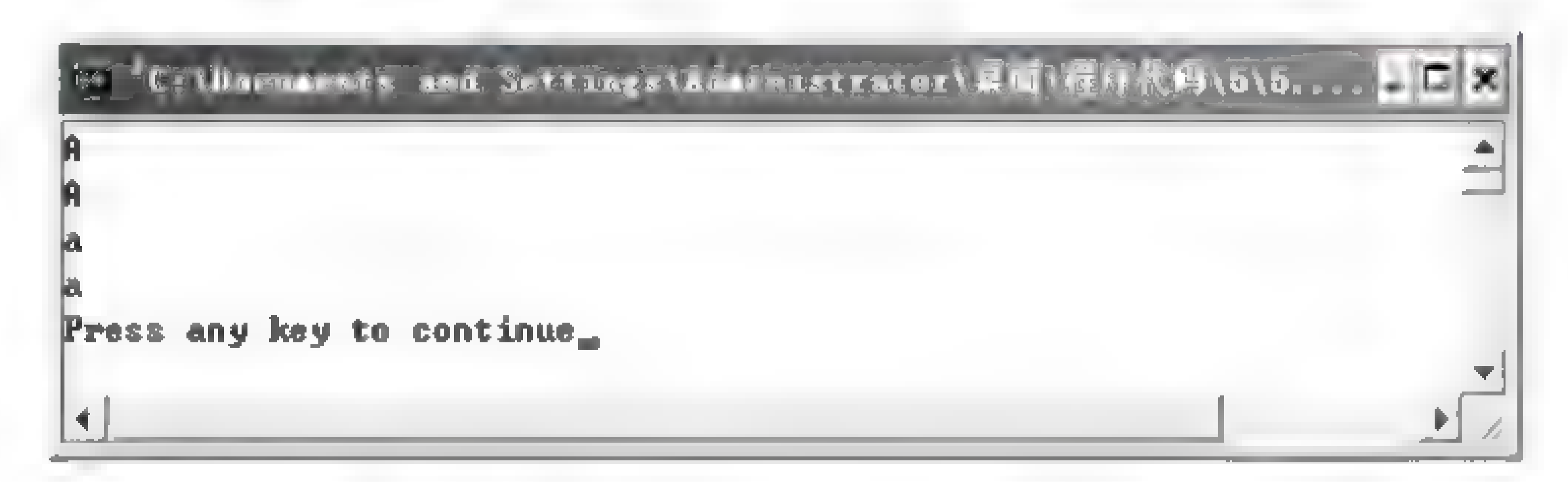


图 5.2　使用 getchar 函数实现字符数据输入

在例 5.2 中，有一处使用 getchar 函数接收回车符，这是怎么回事呢？原来在输入时，当输入完 A 字符后，为了确定输入完毕，要按 Enter 键。回车符也算是一种字符，如果这里不进行获取，那么下次使用 getchar 函数时将得到回车符。下面来看一下不调用 getchar 函数获取回车符的情况。

【例 5.3】　使用 getchar 函数实现字符数据输入（取消获取回车符）。（**实例位置：资源包\TM\sl\5\3**）

```
#include<stdio.h>

int main()
{
    char cChar1;                                /*声明变量*/
    cChar1=getchar();                           /*在输入设备得到字符*/
    putchar(cChar1);                            /*输出字符*/
    putchar('\n');                              /*输出转义字符换行*/
                                                /*将此处 getchar 函数删掉*/
    putchar(getchar());                         /*得到输入字符，直接输出*/
    putchar('\n');                              /*换行*/
    return 0;                                   /*程序结束*/
}
```

这里将 getchar 函数获取回车符的语句去掉了。运行程序，显示效果如图 5.3 所示。



图 5.3　使用 getchar 函数（取消获取回车符）

比较两个程序的运行情况，从程序的显示结果可以发现，程序没有获取第二次的字符输入，而是进行了两次回车操作。

视频讲解

5.3　字符串输入/输出

putchar 和 getchar 函数都只能对一个字符进行操作，如果要进行一个字符串的操作则会很麻烦。C 语言提供了两个对字符串进行操作的函数，分别为 gets 和 puts 函数。

5.3.1　字符串输出函数

字符串输出使用的是 puts 函数，作用是输出一个字符串到屏幕上。其语法格式如下：

```
int puts(char *str);
```

使用 puts 函数时，先要在程序中添加 stdio.h 头文件。其中，形式参数 str 是字符指针类型，可以用来接收要输出的字符串。例如，使用 puts 函数输出一个字符串：

```
puts("I LOVE CHINA!");                              /*输出一个字符串常量*/
```

上述语句首先会输出一个字符串，之后会自动进行换行操作。这与 printf 函数有所不同，在前面的实例中使用 printf 函数进行换行时，要在其中添加转义字符“\n”。puts 函数会在字符串中判断“\0”结束符，遇到结束符时，后面的字符不再输出，并且自动换行。例如：

```
puts("I LOVE\0 CHINA!");                            /*输出一个字符串常量*/
```

在上面的语句中，加上“\0”字符后，puts 函数输出的字符串就变成了“I LOVE”。

说明

前面的章节曾经介绍到，编译器会在字符串常量的末尾添加结束符“\0”，这也就说明了 puts 函数会在输出字符串常量时最后进行换行操作的原因。

【例 5.4】　使用字符串输出函数显示信息提示。（**实例位置：资源包\TM\sl\5\4**）

在本实例中，使用 puts 函数对字符串常量和字符串变量进行操作，在这些操作中观察 puts 函数的

使用方式。

```
#include<stdio.h>

int main()
{
    char* Char="ILOVECHINA";                    /*定义字符串指针变量*/

    puts("ILOVECHINA!");                        /*输出字符串常量*/
    puts("I\0LOVE\0CHINA!");                    /*输出字符串常量，其中加入结束符“\0”*/
    puts(Char);                                 /*输出字符串变量的值*/
    Char="ILOVE\0CHINA!";                       /*改变字符串变量的值*/
    puts(Char);                                 /*输出字符串变量的值*/
    return 0;                                   /*程序结束*/
}
```

（1）字符串常量赋值给字符串指针变量。有关字符串指针的内容将会在后面的章节进行介绍，此时可以将其看作整型变量。为其赋值后，就可以使用该变量。

（2）第一次使用 puts 函数输出的字符串常量中，由于在该字符串中没有结束符“\0”，所以会完整输出整个字符串，直到最后编译器为其添加结束符“\0”为止。

（3）第二次使用 puts 函数输出的字符串常量中，人为添加了两个“\0”，因此只能输出第一个结束符之前的字符，然后进行换行操作。

（4）第三次使用 puts 函数输出的是字符串指针变量，函数根据变量的值进行输出。因为在变量的值中没有结束符，所以会完整输出整个字符串，直至最后编译器为其添加结束字符，然后进行换行操作。

（5）改变变量的值，在使用 puts 函数输出变量时，由于变量的值中包含结束符“\0”，因此将输出第一个结束符后之前的所有字符，然后进行换行操作。

运行程序，显示效果如图 5.4 所示。

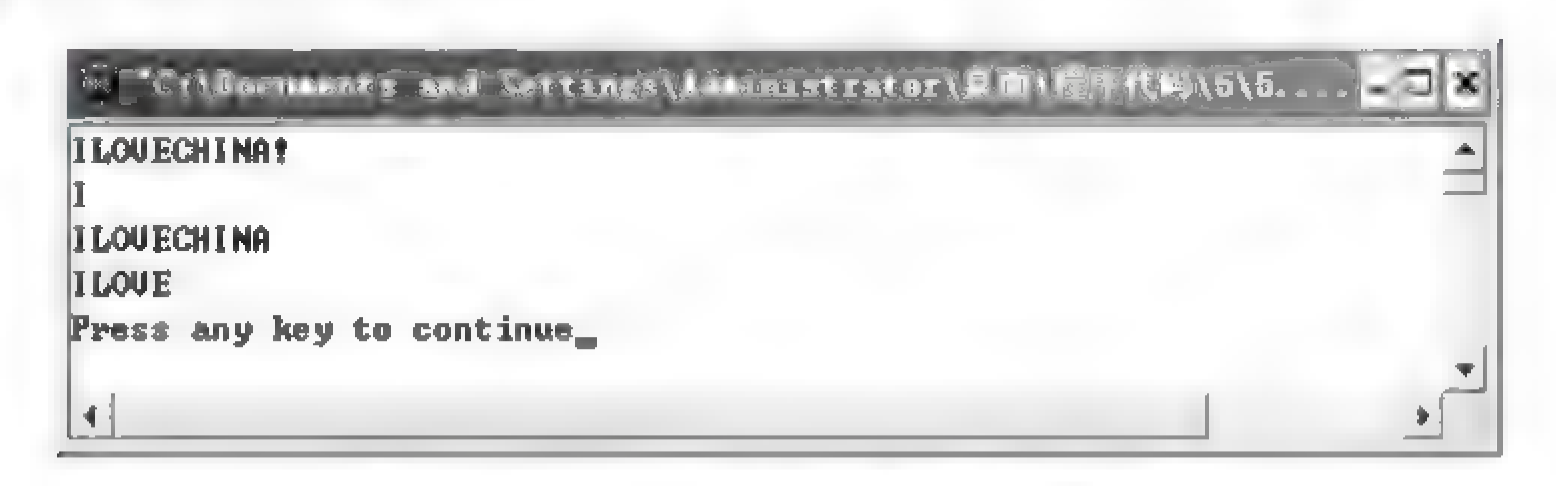


图 5.4　使用字符串输出函数显示信息提示

5.3.2 字符串输入函数

字符串输入使用的是 gets 函数，作用是将读取的字符串保存在形式参数 str 变量中，读取过程直到出现新的一行为止。其中新一行的换行字符将会转换为空终止符“\0”。gets 函数的语法格式如下：

```
char *gets(char *str);
```

在使用 gets 函数输入字符串前，要为程序加入头文件 stdio.h。其中，str 字符指针变量为形式参数。

例如，定义字符数组变量 cString，然后使用 gets 函数获取输入字符的代码如下：

```
gets(cString);
```

在上面的代码中，cString 变量获取了字符串，并将最后的换行符转换成了终止字符。

【例 5.5】　使用字符串输入函数 gets 获取输入信息。（**实例位置：资源包\TM\sl\5\5**）

```
#include<stdio.h>

int main()
{
    char cString[30];                    /*定义一个字符数组变量*/
    gets(cString);                       /*获取字符串*/
    puts(cString);                       /*输出字符串*/
    return 0;                            /*程序结束*/
}
```

（1）因为要接收输入的字符串，所以要定义一个可以接收字符串的变量。在程序代码中，定义 cString 为字符数组变量的标识符。关于字符数组的内容将在后面的章节中进行介绍，此处知道此变量可以接收字符串即可。

（2）调用 gets 函数，其中函数的参数为定义的 cString 变量。调用该函数后，程序会等待用户输入字符，当用户字符输入完毕并按 Enter 键确定时，gets 函数获取字符结束。

（3）使用 puts 字符串输出函数，将获取后的字符串进行输出。

运行程序，显示效果如图 5.5 所示。

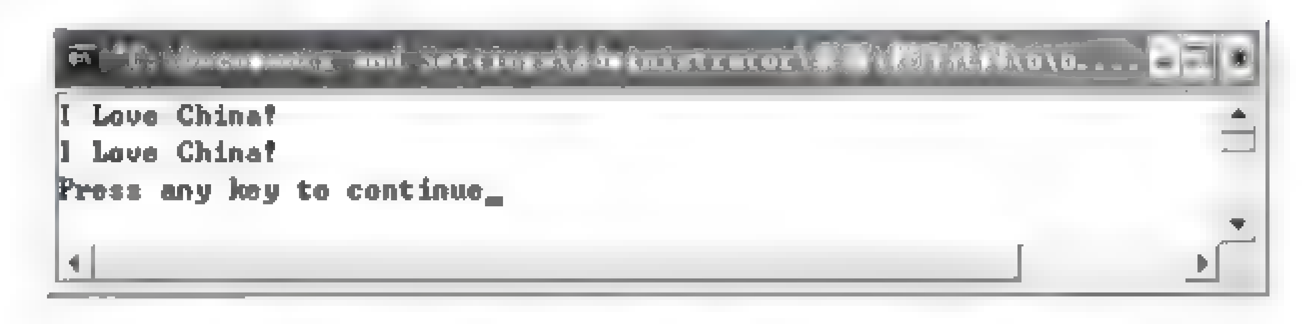


图 5.5　使用字符串输入函数 gets 获取输入信息

5.4　格式输出函数



前面章节的实例中常常使用格式输入/输出函数 scanf 和 printf。其中，printf 函数就是用于格式输出的函数，也称为格式输出函数。

printf 函数的作用是向终端（输出设备）输出若干任意类型的数据，其语法格式如下：

```
printf(格式控制,输出列表)
```

1．格式控制

格式控制是用双引号括起来的字符串，也称为转换控制字符串。其中包括格式字符和普通字符。

☑ 格式字符用来进行格式说明，作用是将输出的数据转换为指定的格式。格式字符通常以“%”字符开头。

☑ 普通字符是需要原样输出的字符，包括双引号内的逗号、空格和换行符。

2. 输出列表

输出列表列出的是要进行输出的一些数据，可以是变量或表达式。

例如，要输出一个整型变量时，代码如下：

```
int iInt=10;
printf("this is %d",iInt);
```

执行上面的语句，显示出来的字符是“this is 10”。格式控制双引号中的字符是“this is %d”，其中的“this is”字符串是普通字符，而“%d”是格式字符，表示输出的是后面的 iInt 数据。

由于 printf 是函数，“格式控制”和“输出列表”这两个位置都是函数的参数，因此 printf 函数的一般形式可以表示为：

```
printf(参数 1,参数 2,…,参数 n)
```

函数中的每一个参数按照给定的格式和顺序依次输出。例如，显示一个字符型变量和整型变量的代码如下：

```
printf("the Int is %d,the Char is %c",iInt,cChar);
```

表 5.1 列出了有关 printf 函数的格式字符。

表 5.1 printf 函数的格式字符

格 式 字 符	功 能 说 明
d,i	以带符号的十进制形式输出整数（整数不输出符号）
o	以八进制无符号形式输出整数
x,X	以十六进制无符号形式输出整数。用 x 输出十六进制数的 a~f 时，以小写形式输出；用 X 时，则以大写字母输出
u	以无符号十进制形式输出整数
c	以字符形式输出，只输出一个字符
s	输出字符串
f	以小数形式输出
e,E	以指数形式输出实数，用 e 时指数以“e”表示，用 E 时指数以“E”表示
g,G	选用“%f”或“%e”格式中输出宽度较短的一种格式，不输出无意义的 0。若以指数形式输出，则指数以大写表示

【例 5.6】 使用格式输出函数 printf 输出不同类型的变量。（**实例位置：资源包\TM\sl\5\6**）

在本实例中，使用 printf 函数对不同类型的变量进行输出，并对使用 printf 函数所用到的输出格式进行分析理解。

```
#include<stdio.h>
int main()
{
    int iInt=10;                                /*定义整型变量*/
    char cChar='A';                             /*定义字符型变量*/
```

```
    float fFloat=12.34f;                           /*定义单精度浮点型*/

    printf("the int is: %d\n",iInt);               /*使用 printf 函数输出整型*/
    printf("the char is: %c\n",cChar);             /*输出字符型*/
    printf("the float is: %f\n",fFloat);           /*输出浮点型*/
    printf("the string is: %s\n","I LOVE YOU");    /*输出字符串*/
    return 0;
}
```

（1）在程序中定义了一个整型变量 iInt，在 printf 函数中使用格式字符“%d”进行输出。

（2）将字符型变量 cChar 赋值为 A，在 printf 函数中使用格式字符“%c”输出字符。

（3）格式字符“%f”用来输出实型变量的数值。

（4）在最后一个 printf 输出函数中，使用“%s”将一个字符串进行输出，字符串不包括双引号。

运行程序，显示效果如图 5.6 所示。

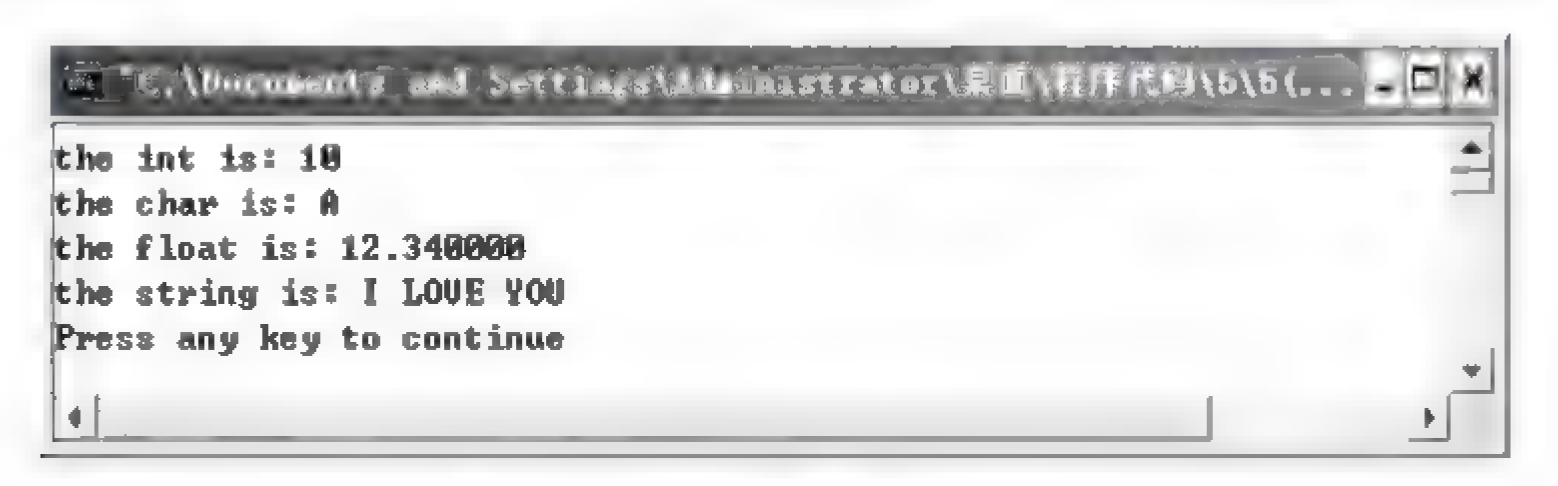


图 5.6　使用格式输出函数 printf

另外，在格式说明中，在“%”符号和上述格式字符间可以插入如表 5.2 所示的几种附加符号。

表 5.2　printf 函数的附加格式说明字符

字　符	功 能 说 明
字母 l	用于长整型整数，可加在格式字符 d、o、x、u 前面
m（代表一个整数）	数据最小宽度
n（代表一个整数）	对实数，表示输出 n 位小数；对字符串，表示截取的字符个数
-	输出的数字或字符在域内向左靠拢

注意

在使用 printf 函数时，除 X、E、G 外，其他格式字符必须使用小写字母，如“%d”不能写成“%D”。

如果想输出“%”符号，则在格式控制处使用“%%”进行输出即可。

【例 5.7】　在 printf 函数中使用附加格式说明字符。（**实例位置：资源包\TM\sl\5\7**）

在本实例中，使用 printf 函数的附加格式说明字符，对输出的数据进行更为精准的格式设计。

```
#include<stdio.h>

int main()
{
    long iLong=100000;                             /*定义长整型变量，为其赋值*/
```

```
    printf("the Long is %ld\n",iLong);                  /*输出长整型变量*/

    printf("the string is: %s\n","LOVE");               /*输出字符串*/
    printf("the string is: %10s\n","LOVE");             /*使用 m 控制输出列*/
    printf("the string is: %-10s\n","LOVE");            /*使用-表示向左靠拢*/
    printf("the string is: %10.3s\n","LOVE");           /*使用 n 表示取字符数*/
    printf("the string is: %-10.3s\n","LOVE");
    return 0;
}
```

（1）在程序代码中，定义的长整型变量在使用 printf 函数对其进行输出时，应该在“%d”格式字符中添加 l 字符，继而输出长整型变量。

（2）“%s”用来输出一个字符串的格式字符，在结果中可以看到输出了字符串“LOVE”。

（3）“%10s”格式为“%ms”，表示输出字符串占 m 列。如果字符串本身长度大于 m，则突破 m 的限制，将字符串全部输出；若字符串长度小于 m，则用空格进行左补齐。可以看到在字符串“LOVE”前后存在 6 个空格。

（4）“%-10s”格式为“%-ms”，表示如果字符串长度小于 m，则在 m 列范围内，字符串向左靠，右补空格。

（5）“%10.3s”格式为“%m.ns”，表示输出占 m 列，但只取字符串左端 n 个字符。这 n 个字符输出在 m 列的右侧，左补空格。

（6）“%-10.3s”格式为“%-m.ns”，其中 m、n 含义同上，n 个字符输出在 m 列范围内的左侧，右补空格。如果 n>m，则 m 自动取 n 值，即保证 n 个字符正常输出。

运行程序，显示效果如图 5.7 所示。

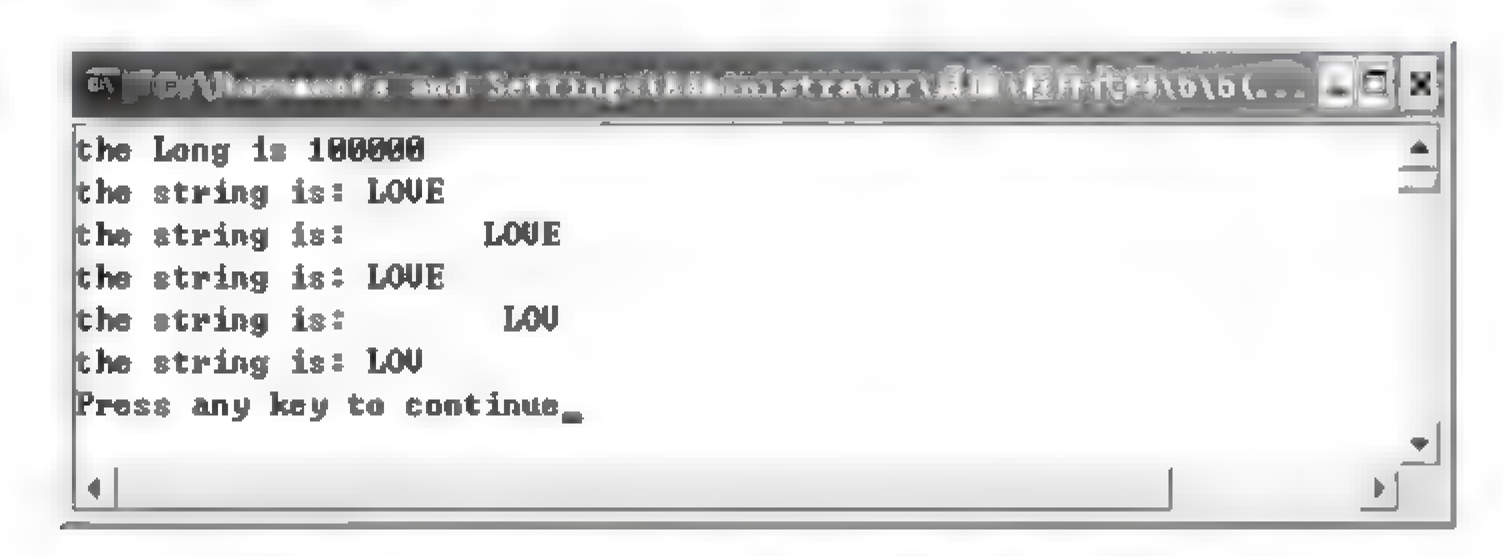


图 5.7 在 printf 函数中使用附加格式说明字符



5.5 格式输入函数

与格式输出函数 printf 相对应的是格式输入函数 scanf。该函数的功能是指定固定的格式，并且按照指定的格式接收用户在键盘上输入的数据，最后将数据存储在指定的变量中。

scanf 函数的一般格式如下：

```
scanf(格式控制,地址列表)
```

通过 scanf 函数的一般格式可以看出，参数位置中的格式控制与 printf 函数相同。如“%d”表示十

进制的整型，“%c” 表示单字符。地址列表中用于给出接收数据变量的地址。例如，得到一个整型数据的代码如下：

```
scanf("%d",&iInt);                                    /*得到一个整型数据*/
```

在上面的代码中，“&” 符号表示取 iInt 变量的地址。用户不用关心变量的地址具体是多少，只要在变量的标识符前加 “&”，就可以表示存取变量的地址。

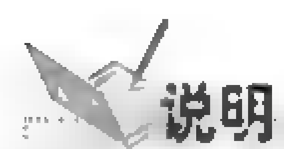
注意

编写程序时，在 scanf 函数参数的地址列表处，一定要使用变量的地址，而不是变量的标识符，否则编译器会提示出现错误。

表 5.3 中列出了 scanf 函数中常用的格式字符。

表 5.3　scanf 函数的格式字符

格 式 字 符	功 能 说 明
d，i	用来输入有符号的十进制整数
u	用来输入无符号的十进制整数
o	用来输入无符号的八进制整数
x，X	用来输入无符号的十六进制整数（大小写作用是相同的）
c	用来输入单个字符
s	用来输入字符串
f	用来输入实型，可以用小数形式或指数形式输入
e，E，g，G	与 f 作用相同，e 与 f、g 之间可以相互替换（大小写作用相同）



说明

格式字符 “%s” 用来输入字符串。将字符串送到一个字符数组中，在输入时以非空白字符开始，以第一个空白字符结束。字符串以串结束标志 “\0” 作为最后一个字符。

【例 5.8】　使用 scanf 格式输入函数得到用户输入的数据。（**实例位置：资源包\TM\sl\5\8**）

在本实例中，利用 scanf 函数得到用户输入的两个整型数据，因为 scanf 函数只能用于输入操作，所以若在屏幕上显示信息，还需要使用显示函数。

```
#include<stdio.h>

int main()
{
    int iInt1,iInt2;                                  /*定义两个整型变量*/
    puts("Please enter two numbers:");                /*通过 puts 函数输出提示信息的字符串*/
    scanf("%d%d",&iInt1,&iInt2);                      /*通过 scanf 函数得到输入的数据*/
    printf("The first is : %d\n",iInt1);              /*显示第一个输入的数据*/
    printf("The second is : %d\n",iInt2);             /*显示第二个输入的数据*/
    return 0;
}
```

（1）为了能接收用户输入的整型数据，定义了两个整型变量 iInt1 和 iInt2。

（2）因为 scanf 函数只能接收用户的数据，而不能显示信息，所以先使用 puts 函数输出一段字符表示信息提示。puts 函数在输出字符串之后会自动进行换行，这样就可以省去使用换行符。

（3）调用 scanf 格式输入函数，在函数参数中可以看到，在格式控制的位置使用双引号将格式字符包括，“%d”表示输入的是十进制的整数。在参数中的地址列表位置，使用“&”符号表示变量的地址。

（4）此时变量 iInt1 和 iInt2 已经得到了用户输入的数据，调用 printf 函数将变量进行输出。这里要注意区分的是，printf 函数使用的是变量的标识符，而不是变量的地址。scanf 函数使用的是变量的地址，而不是标识符。


说明

程序是怎样将输入的内容分别保存到指定的两个变量中的呢？原来 scanf 函数使用空白字符分隔输入的数据，这些空白字符包括空格、换行、制表符（tab）。例如在本程序中，使用换行作为空白字符。

运行程序，显示效果如图 5.8 所示。

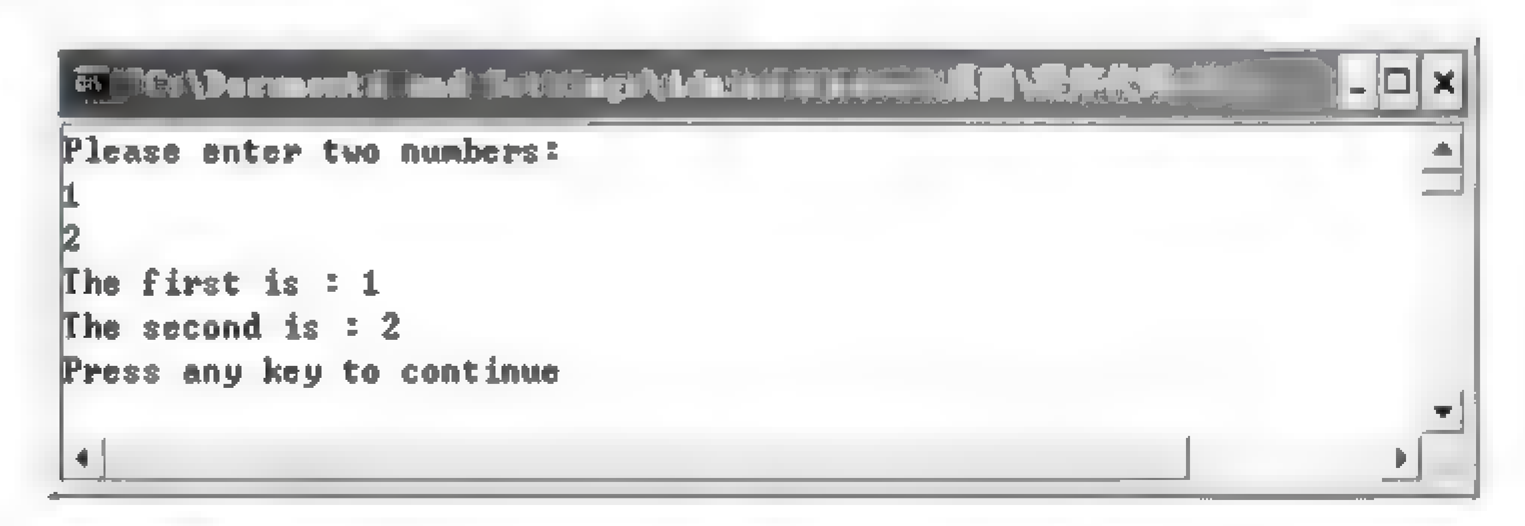


图 5.8　使用 scanf 格式输入函数得到用户输入的数据

在 printf 函数中，附加格式用于更为具体的说明。相应地，scanf 函数中也有附加格式，用于更为具体的格式说明，如表 5.4 所示。

表 5.4　scanf 函数的附加格式

字　符	功 能 说 明
l	用于输入长整型数据（可用于“%ld”“%lo”“%lx”“%lu”）以及 double 型的数据（“%lf”或“%le”）
h	用于输入短整型数据（可用于“%hd”“%ho”“%hx”）
n（整数）	指定输入数据所占的宽度
*	表示指定的输入项在读入后不赋给相应的变量

【例 5.9】　使用 scanf 函数的附加格式进行格式输入。（**实例位置：资源包\TM\sl\5\9**）

在本实例中，依次使用 scanf 函数的附加格式进行格式输入，对比输入前后的结果，观察附加格式的效果。

```
#include<stdio.h>

int main()
```

```
{
    long iLong;                                                    /*长整型变量*/
    short iShort;                                                  /*短整型变量*/
    int iNumber1=1;                                                /*整型变量，为其赋值为 1*/
    int iNumber2=2;                                                /*整型变量，为其赋值为 2*/
    char cChar[10];                                                /*定义字符数组变量*/

    printf("Enter the long integer\n");                            /*输出信息提示*/
    scanf("%ld",&iLong);                                           /*输入长整型数据*/

    printf("Enter the short integer\n");                           /*输出信息提示*/
    scanf("%hd",&iShort);                                          /*输入短整型数据*/

    printf("Enter the number:\n");                                 /*输出信息提示*/
    scanf("%d*%d",&iNumber1,&iNumber2);                            /*输入整型数据*/

    printf("Enter the string but only show three character\n");    /*输出信息提示*/
    scanf("%3s",cChar);                                            /*输入字符串*/

    printf("the long interger is: %ld\n",iLong);                   /*显示长整型值*/
    printf("the short interger is: %hd\n",iShort);                 /*显示短整型值*/
    printf("the Number1 is: %d\n",iNumber1);                       /*显示整型 iNumber1 的值*/
    printf("the Number2 is: %d\n",iNumber2);                       /*显示整型 iNumber2 的值*/
    printf("the three character are: %s\n",cChar);                 /*显示字符串*/
    return 0;
}
```

（1）为了 scanf 函数能接收数据，在程序代码中定义所使用的变量。为了演示不同格式说明的情况，定义的变量类型有长整型、短整型和字符数组。

（2）使用 printf 函数显示一串字符，提示输入的数据为长整型，调用 scanf 函数使变量 iLong 得到用户输入的数据。在 scanf 函数的格式控制部分，使用附加格式字符 l 表示长整型。

（3）再使用 printf 函数显示数据提示，提示输入的数据为短整型。调用 scanf 函数时，使用附加格式字符 h 表示短整型。

（4）使用格式字符“*”的作用是表示指定的输入项在读入后不赋给相应的变量，在代码中分析这句话的含义就是，第一个“%d”是输入 iNumber1 变量，第二个“%d”是输入 iNumber2 变量，但是在第二个“%d”前有一个“*”附加格式说明字符，这样第二个输入的值被忽略，也就是说，iNumber2 变量不保存输入相应的值。

（5）“%s”是用来表示字符串的格式字符，将一个数 n（整数）放入“%s”中间，这样就指定了数据的宽度。在程序中，scanf 函数中指定的数据宽度为 3，那么在输入一个字符串时，只是接收前 3 个字符。

（6）利用 printf 函数将输入得到的数据进行输出。

运行程序，显示效果如图 5.9 所示。

```
Enter the long integer
100000
Enter the short integer
1[redacted]
Enter the number:
1[redacted]
Enter the string but only show three character
Wonderful
the long interger is: 100000
the short interger is: 10000
the Number1 is: 10
the Number2 is: 2
the three character are: Won
Press any key to continue
```

图 5.9　使用附加格式说明 scanf 函数的格式输入



5.6　顺序程序设计应用

本节将介绍几个顺序程序设计的实例，帮助读者巩固前面所讲的内容。

【例 5.10】　计算圆的面积。（**实例位置：资源包\TM\sl\5\10**）

在本实例中，定义单精度浮点型变量，为其赋值为圆周率的值。得到用户输入的数据并进行计算，最后将计算的结果输出。

```
#include<stdio.h>

int main()
{
    float Pie=3.14f;                          /*定义圆周率*/

    float fArea;                              /*定义变量，表示圆的面积*/
    float fRadius;                            /*定义变量，表示圆的半径*/

    puts("Enter the radius:");                /*输出提示信息*/
    scanf("%f",&fRadius);                     /*输入圆的半径*/
    fArea=fRadius*fRadius*Pie;                /*计算圆的面积*/
    printf("The Area is: %.2f\n",fArea);      /*输出计算的结果*/
    return 0;                                 /*程序结束*/
}
```

（1）定义单精度浮点型变量 Pie 表示圆周率（在常量 3.14 后加上 f 表示为单精度类型），变量 fArea 表示圆的面积，变量 fRadius 表示圆的半径。

（2）根据 puts 函数输出的程序提示信息，使用 scanf 函数输入半径的数据，将输入的数据保存在变量 fRadius 中。

（3）圆的面积=圆的半径的平方×圆周率。运用公式，将变量放入其中计算圆的面积，最后使用 printf 函数将结果输出。在 printf 函数中可以看到“%.2f”格式关键字，其中的“.2”表示取小数点后的两位。

运行程序，显示效果如图 5.10 所示。



图 5.10　计算圆的面积

【例 5.11】　将大写字符转换成小写字符。（**实例位置：资源包\TM\sl\5\11**）

本实例要将输入的大写字符转换成小写字符。要解决这个问题，就需要对 C 语言中的 ASCII 码有所了解。将大写字符转换成小写字符的方法就是将大写字符的 ASCII 码转换成小写字符的 ASCII 码。

```
#include<stdio.h>

int main()
{
    char cBig;                                          /*定义字符变量，表示大写字符*/
    char cSmall;                                        /*定义字符变量，表示小写字符*/

    puts("Please enter capital character:");            /*输出提示信息*/
    cBig=getchar();                                     /*得到用户输入的大写字符*/
    puts("Minuscule character is:");                    /*输出提示信息*/
    cSmall=cBig+32;                                     /*将大写字符转换成小写字符*/
    printf("%c\n",cSmall);                              /*输出小写字符*/
    return 0;                                           /*程序结束*/
}
```

（1）为了将大写字符转换为小写字符，要为其定义变量并进行保存。cBig 表示要输入的大写字符变量，而 cSmall 表示要转换成的小写字符变量。

（2）通过信息提示，用户输入字符。因为只要得到一个输入的字符即可，所以在此处使用 getchar 函数就可以满足程序的要求。

（3）大写字符与小写字符的 ASCII 码值相差 32。例如，字符 A 的 ASCII 值为 65，a 的 ASCII 值为 97。因此要将一个大写字符转换成小写字符，将大写字符的 ASCII 值加上 32 即可。

（4）字符变量 cSmall 得到转换的小写字符后，利用 printf 格式输出函数将字符输出，其中使用的格式字符为“%c”。

运行程序，显示效果如图 5.11 所示。

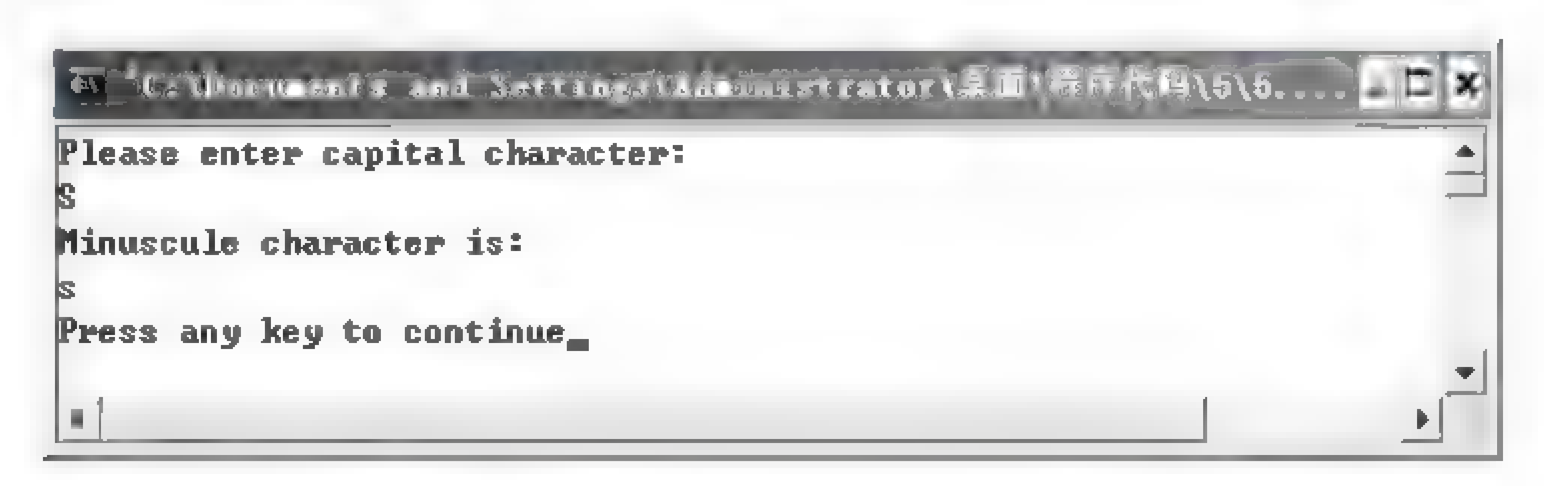


图 5.11　将大写字符转换成小写字符

5.7 小　　结

本章主要讲解C语言中常用的数据输入、输出函数。熟练使用输入、输出函数是学习C语言必须要掌握的，因为在很多情况下，为了证实一项操作的正确性，可以将输入和输出的数据进行对比而得到结论。

其中，用于单个字符的输入和输出时，使用的是getchar和putchar函数。gets和puts函数用于输入和输出字符串，并且puts函数在遇到终止符时会进行自动换行。为了能输出其他类型的数据，可以使用格式输出函数printf和格式输入函数scanf。在这两个格式函数中，利用格式字符和附加格式字符可以更为具体地进行格式说明。

5.8 实践与练习

1. 模仿例5.11，将输入的小写字符转换成大写字符，并且将与大写字符相对应的ASCII码进行输出。**（答案位置：资源包\TM\sl\5\12）**

2. 模拟工资计算器，计算一个销售人员的月工资数量（月工资=基本工资+提成，提成=商品数×1.5）。**（答案位置：资源包\TM\sl\5\13）**

第 6 章

选择结构程序设计

（视频讲解：39 分钟）

步入程序设计领域的第一步，是学会设计编写一个程序。顺序结构程序设计最简单，与其相比，选择结构程序设计还需要用到一些用于条件判断的语句，程序的功能更加复杂，程序的逻辑性与灵活性也更加强大。

本章致力于使读者掌握使用 if 语句进行条件判断的方法，并掌握 switch 语句的使用方式。

通过阅读本章，您可以：

- 使用 if 语句编写判断语句
- 掌握 switch 语句的编写方式
- 区分两种 if...else 语句与 switch 语句
- 通过应用程序了解选择结构的具体使用方法



6.1　if 语句

在日常生活中，为了使交通畅通有序，一般会在路口设立交通信号灯。在信号灯显示为绿色时，车辆可以行驶通过；当信号灯转为红色时，车辆就要停止行驶。可见，信号灯给出了信号，人们通过不同的信号进行判断，然后根据判断的结果进行相应的操作。

在 C 语言程序中，要想完成这样的判断操作，利用的就是 if 语句。if 语句的功能就像路口的信号灯一样，通过判断不同的条件，决定是否进行操作。

据说第一台数字计算机是用来进行决策操作的，使得之后的计算机都继承了这项功能。程序员将决策表示成对条件的检验，即判断一个表达式值的真假。除了没有任何返回值的函数和返回无法判断真假的结构函数外，几乎所有表达式的返回值都可以判断真假。

下面介绍 if 语句的有关具体内容。



6.2　if 语句的基本形式

在 if 语句中，首先判断表达式的值，然后根据该值的情况控制程序流程。表达式的值不等于 0，也就是为真；否则，就是假值。if 语句有 if、if...else 和 else if 3 种形式，下面介绍每种形式的具体使用方式。

6.2.1　if 语句形式

if 语句通过对表达式进行判断，根据判断的结果决定是否进行相应的操作。if 语句的一般形式如下：

```
if(表达式)  语句
```

其语句的执行流程图如图 6.1 所示。

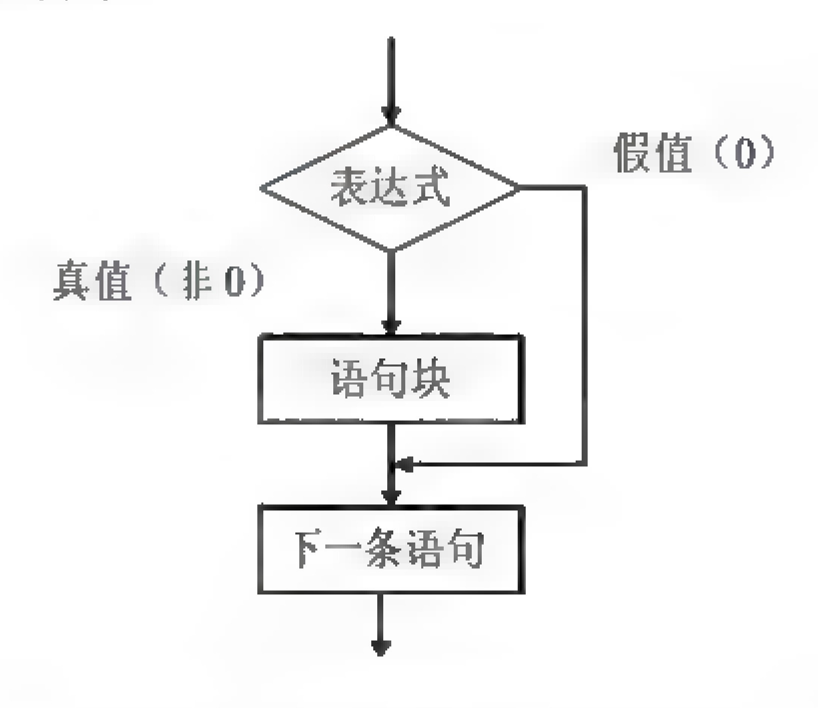


图 6.1　if 语句的执行流程图

if 后面括号中的表达式就是要进行判断的条件，后面的语句部分则是对应的操作。如果 if 判断括号中的表达式为真，就执行后面语句的操作；如果为假值，那么不会执行后面的语句部分。例如下面

的代码：

```
if(iNum) printf("The truevalue");
```

代码中判断变量 iNum 的值，如果变量 iNum 为真值，则执行后面的输入语句；如果变量的值为假，则不执行后面的语句。

在 if 语句的括号中，不仅可以判断一个变量的值是否为真，也可以判断一个表达式的值是否为真，例如：

```
if(iSignal==1) printf("the Signal Light is%d:",iSignal);
```

这行代码的含义是：判断变量 iSignal==1 的表达式，如果条件成立，那么判断的结果是真值，则执行后面的输出语句；如果条件不成立，那么结果为假值，则不执行后面的输出语句。

上述示例代码中，if 后面的执行部分只是一条语句。如果有两条语句，又该怎么办呢？这时可以使用大括号，使之成为语句块。例如：

```
if(iSignal==1)
{
    printf("the Signal Light is%d:\n",iSignal);
    printf("Cars can run");
}
```

将执行的语句都放在大括号中，这样当 if 语句判断条件为真时，语句块内的内容将会全部被执行。使用这种方式的好处是可以规范、清晰地表达出 if 语句所包含的范围。建议大家在使用 if 语句时，都使用大括号将执行语句包括在内。

【例 6.1】 使用 if 语句模拟信号灯，指挥车辆行驶。（**实例位置：资源包\TM\sl\6\1**）

在本实例中，为了模拟十字路口的信号灯指挥系统，要使用 if 语句判断信号灯的状态。如果信号灯为绿色，则说明车辆可以行驶通过，通过输出语句进行信息提示，说明车辆的行动状态。

```
#include<stdio.h>

int main()
{
    int iSignal;                                          /*定义变量表示信号灯的状态*/
    printf("the Red Light is 0,the Green Light is 1\n");  /*输出提示信息*/
    scanf("%d",&iSignal);                                 /*输入 iSignal 变量*/
    if(iSignal==1)                                        /*使用 if 语句进行判断*/
    {
        printf("the Light is green,cars can run\n");      /*判断结果为真时输出*/
    }
    return 0;
}
```

（1）为了模拟信号灯指挥系统，要根据信号灯的状态进行判断，这样就需要一个变量表示信号灯的状态。在程序代码中，定义变量 iSignal 表示信号灯的状态。

（2）输出提示信息，输入 iSignal 变量，表示此时信号灯的状态。此时用键盘输入“1”，表示信号灯的状态是绿灯。

（3）使用 if 语句判断 iSignal 变量的值，如果为真，则表示信号灯为绿灯；如果为假，则表示信号灯是红灯。在程序中，此时变量 iSignal 的值为 1，表达式 iSignal==1 的条件成立，因此判断的结果为真值，从而执行 if 语句后面大括号中的语句。

运行程序，显示效果如图 6.2 所示。

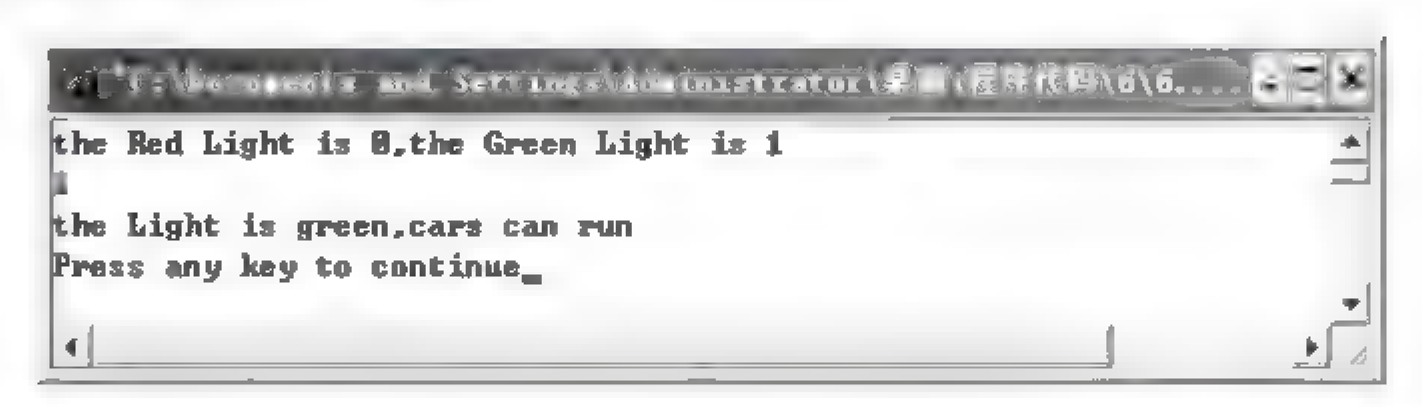


图 6.2　使用 if 语句模拟信号灯指挥车辆行驶

if 语句可以使用多次，通过连续使用进行进一步的判断，继而根据不同的分支条件给出相应的操作。

例如在例 6.1 中，虽然使用 if 语句判断了信号灯状态 iSignal 变量，但是只给出了判断是绿灯时执行的操作，并没有给出红灯时相应的操作。为了使得在红灯情况下也能进行操作，需要再使用一次 if 语句。现在对例 6.1 进行完善。

【例 6.2】　完善 if 语句的使用。（**实例位置：资源包\TM\sl\6\2**）

例 6.1 仅对绿灯情况做出了相应操作，为进一步完善信号灯为红灯时的操作，在程序中再添加一次 if 语句，对信号灯为红灯的情况进行判断，并且在条件成立时给出相应操作。

```
#include<stdio.h>

int main()
{
	int iSignal;                                        /*定义变量表示信号灯的状态*/
	printf("the Red Light is 0,the Green Light is 1\n");  /*输出提示信息*/
	scanf("%d",&iSignal);                               /*输入 iSignal 变量*/

	if(iSignal==1)                                      /*使用 if 语句进行判断*/
	{
		printf("the Light is green,cars can run\n");    /*判断结果为真时输出*/
	}
	if(iSignal==0)                                      /*使用 if 语句进行判断*/
	{
		printf("the Light is red,cars can't run\n");    /*判断结果为真时输出*/
	}
	return 0;
}
```

（1）在例 6.1 的基础上进行修改，完善程序的功能。在代码中添加一个 if 判断语句，用来表示当信号灯为红灯时所进行的相应操作。

（2）从程序的开始处来分析整个程序的运行过程。使用 scanf 函数输入数据，这次用户输入的是“0”，表示红灯。

（3）程序继续执行，第一个 if 语句判断 iSignal 变量的值是否为 1，如果判断的结果为真，则说明信号灯为绿灯。因为 iSignal 变量的值为 0，所以判断的结果为假，不会执行后面语句中的内容。

（4）接下来是新添加的 if 语句，在其中判断 iSignal 变量是否等于 0，如果判断成立为真，则表示信号灯此时为红灯。因为输入的值为 0，所以 iSignal==0 条件成立，执行 if 后面的语句内容。

运行程序，显示效果如图 6.3 所示。



图 6.3　完善 if 语句的使用

初学编程的人在程序中使用 if 语句时，常常会将下面的两个判断弄混，例如：

```
if(value){...}                                  /*判断变量值*/
if(value==0){...}                               /*判断表达式的值*/
```

这两行代码中都有 value 变量，value 值虽然相同，但是判断的结果却不同。第 1 行代码判断的是 value 的值，第 2 行代码判断的是 value 等于 0 这个表达式是否成立。假定其中 value 的值为 0，那么在第一个 if 语句中，value 值为 0 即说明判断的结果为假，所以不会执行 if 后的语句。在第二个 if 语句中，判断的是 value 是否等于 0，因为设定 value 的值为 0，所以表达式成立，那么判断的结果就为真，执行 if 后的语句。

6.2.2　if...else 语句形式

除了可以指定在条件为真时执行某些语句外，还可以在条件为假时执行另外一段代码。这在 C 语言中是利用 else 语句来完成的，其一般形式如下：

```
if(表达式)
    语句块 1;
else
    语句块 2;
```

其语句的执行流程图如图 6.4 所示。

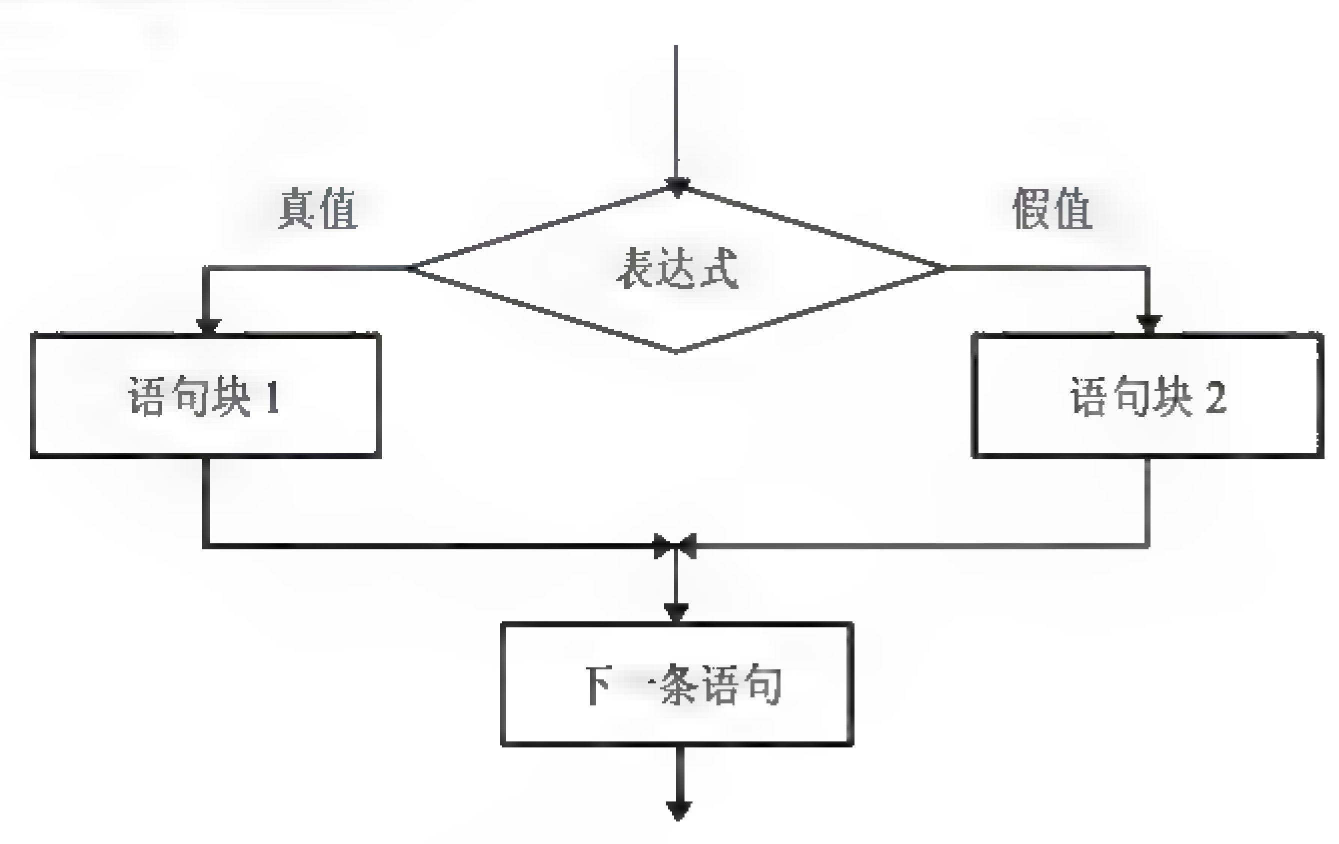


图 6.4　if...else 语句的执行流程图

在 if 后的括号中判断表达式的结果，如果判断的结果为真值，则执行紧跟 if 后的语句块中的内容；如果判断的结果为假值，则执行 else 语句后的语句块内容。也就是说，当 if 语句检验的条件为假时，就执行相应的 else 语句后面的语句或者语句块。例如：

```
if(value)
{
    printf("the value is true");
}
else
{
    printf("the value is false");
}
```

在上面的代码中，如果 if 判断变量 value 的值为真，则执行 if 后面的语句块。如果 if 判断的结果为假值，则执行 else 下面的语句块。

注意

一个 else 语句必须跟在一个 if 语句的后面。

【例 6.3】 使用 if...else 语句进行选择判断。（**实例位置：资源包\TM\sl\6\3**）

在本实例中，使用 if...else 语句判断用户输入的数值。输入的数字为 0，表示条件为假；输入的数字为非 0，表示条件为真。

```
#include<stdio.h>

int main()
{
    int iNumber;                                    /*定义变量*/

    printf("Enter a number\n");                     /*显示提示信息*/
    scanf("%d",&iNumber);                           /*输入数字*/

    if(iNumber)                                     /*判断变量的值*/
    {
                                                    /*判断为真时执行输出*/
        printf("the value is true and the number is: %d\n",iNumber);
    }
    else                                            /*判断为假时执行输出*/
    {
        printf("the value is flase and the number is: %d\n",iNumber);
    }
    return 0;
}
```

（1）程序中定义变量 iNumber 用来保存用户输入的数据，然后通过 if...else 语句判断变量的值。

（2）用户输入数据的值为 0，if 语句判断 iNumber 变量，此时也就是判断输入的数值。因为 0 表

示的是假，所以不会执行 if 后面紧跟着的语句块，而会执行 else 后面语句块中的操作，显示一条信息并将数值进行输出。

（3）从程序的运行结果也可以看出，当 if 语句检验的条件为假时，就执行相应的 else 语句后面的语句或者语句块。

运行程序，显示效果如图 6.5 所示。

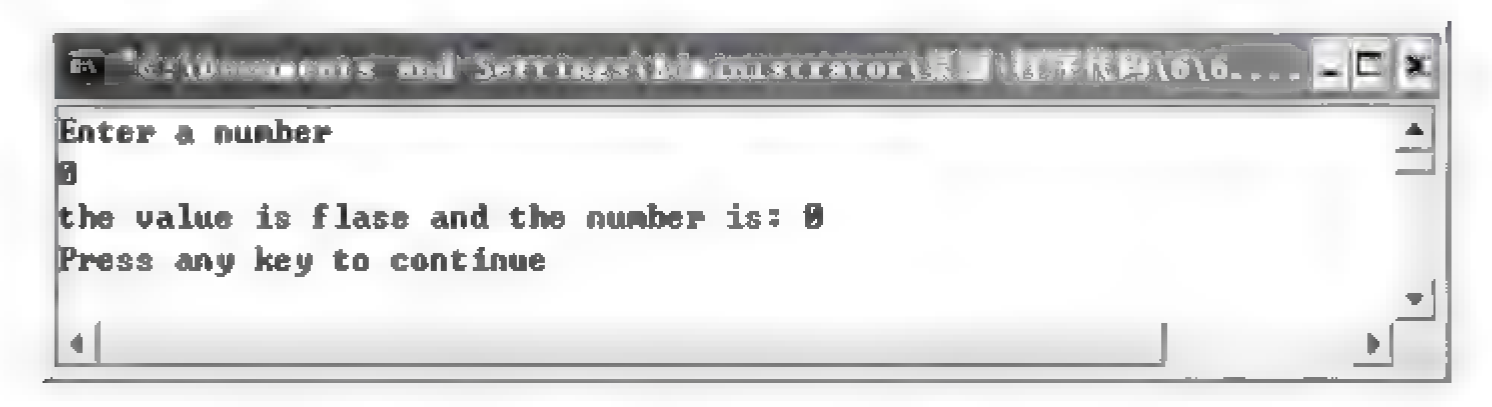


图 6.5　使用 if...else 语句进行选择判断

if...else 语句也可以用来判断表达式，根据表达式的结果选择不同的操作。

【例 6.4】　使用 if...else 语句得到两个数中的最大值。（**实例位置：资源包\TM\sl\6\4**）

本实例要实现的功能是比较两个数值的大小。这两个数值由用户输入，然后将其中相对较大的数值输出显示。

```
#include<stdio.h>

int main()
{
	int iNumber1,iNumber2;                          /*定义变量*/

	printf("please enter two numbers:\n");          /*信息提示*/
	scanf("%d%d",&iNumber1,&iNumber2);              /*输入数据*/
	if(iNumber1>iNumber2)                           /*判断 iNumber1 是否大于 iNumber2*/
	{
		printf("the bigger number is %d\n",iNumber1);
	}
	else                                            /*判断结果为假，则执行下面的语句*/
	{
		printf("the bigger number is %d\n",iNumber2);
	}
	return 0;
}
```

（1）在程序运行过程中，利用 printf 函数先显示一条信息，通过信息提示用户输入两个数据，第一个输入的是 5，第二个输入的是 10。这两个数据的数值由变量 iNumber1 和 iNumber2 保存。

（2）if 语句判断表达式 iNumber1>iNumber2 的真假。如果判断的结果为真，则执行 if 后的语句，输出 iNumber1 的值，说明 iNumber1 是最大值；如果判断的结果为假，则执行 else 后的语句，输出 iNumber2 的值，说明 iNumber2 是最大值。因为 iNumber1 的值为 5，iNumber2 的值为 10，所以 iNumber1>iNumber2 的关系表达式结果为假，这样执行的就是 else 后的语句，输出 iNumber2 的值。

运行程序，显示效果如图 6.6 所示。

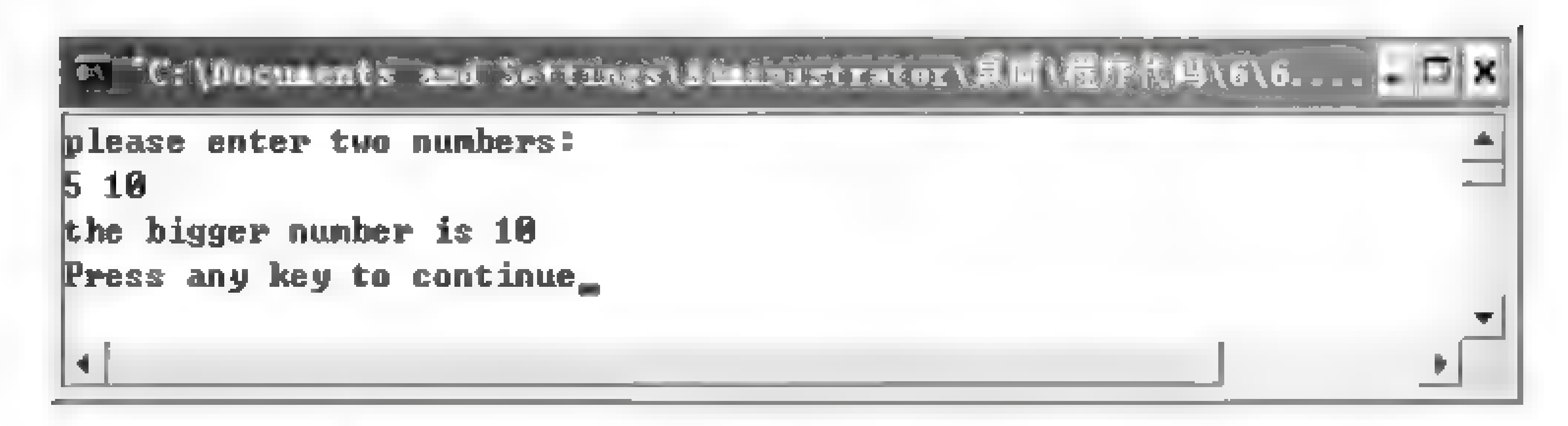


图 6.6　使用 if...else 语句得到两个数的最大值

【例 6.5】　使用 if...else 语句模拟信号灯。（**实例位置：资源包\TM\sl\6\5**）

多数路口的信号灯系统中，除了红灯、绿灯外，还有一个黄灯，用来提示车辆准备行驶或者停车。6.2.1 节使用 if 语句模拟信号灯，在本实例中使用 if...else 语句来进一步完善这个程序，使得信号灯具有黄灯相应的功能。

```
#include<stdio.h>

int main()
{
    int iSignal;                                          /*定义变量表示信号灯的状态*/
    printf("the Red Light is 0,\nthe Green Light is 1,\nthe Yellow Light is other number\n");    /*输出提示信息*/
    scanf("%d",&iSignal);                                 /*输入 iSignal 变量*/

    if(iSignal==1)                                        /*当信号灯为绿灯时*/
    {
        printf("the Light is green,cars can run\n");      /*判断结果为真时输出*/
    }
    if(iSignal==0)                                        /*当信号灯为红灯时*/
    {
        printf("the Light is red,cars can't run\n");      /*判断结果为真时输出*/
    }
    else                                                  /*当信号灯为黄灯时*/
    {
        printf("the Light is yellow,cars are ready\n");
    }
    return 0;
}
```

（1）程序运行时，先输出信息，提示用户输入一个信号灯的状态。其中，0 表示红灯，1 表示绿灯，其他数字表示黄灯。

（2）输入一个数字 2，将其保存到变量 iSignal 中。接下来使用 if 语句进行判断。

（3）第一个 if 语句判断 iSignal 是否等于 1，很明显判断结果为假，因此不会执行第一个 if 后的语句块中的内容。

（4）第二个 if 语句判断 iSignal 是否等于 0，结果为假，因此不会执行第二个 if 后的语句块中的内容。

（5）因为第二个 if 语句为假值，不执行第二个 if 语句的话，就会执行 else 后的语句块。在语句块中通过输出信息表示现在为黄灯，车辆要进行准备。

运行程序，显示效果如图 6.7 所示。

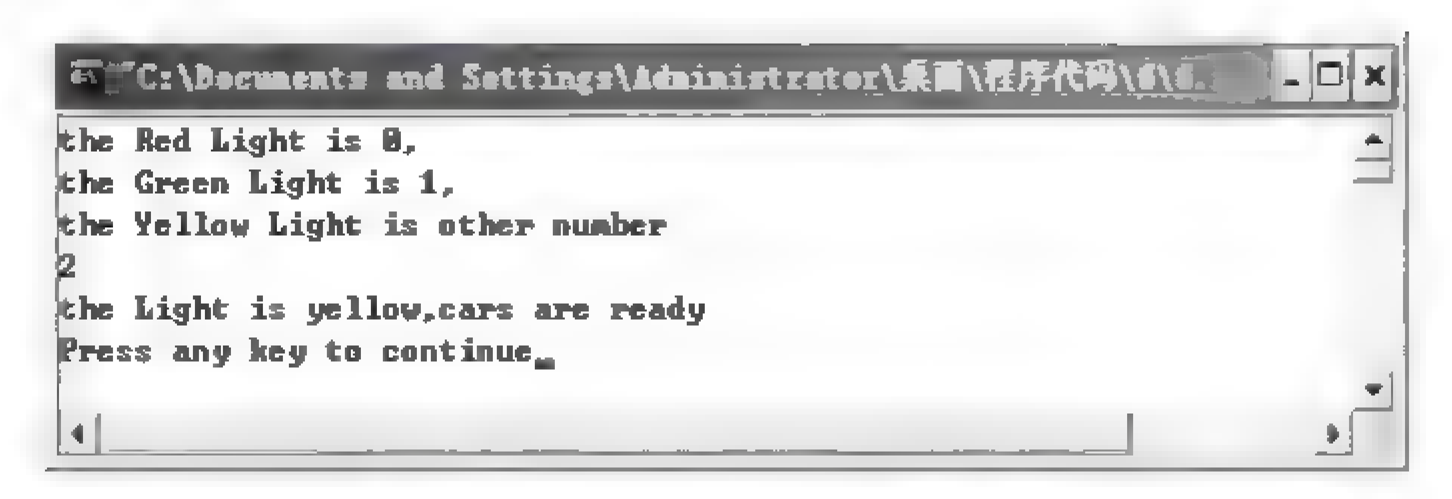

```
the Red Light is 0,
the Green Light is 1,
the Yellow Light is other number
2
the Light is yellow,cars are ready
Press any key to continue
```

图 6.7　使用 if...else 语句模拟信号灯

注意

上面这个程序实际上是存在一些问题的，假如用户输入的数值为 1，第一个 if 语句判断为真值，则会执行后面紧跟着的语句块。并且因为第二个 if 语句判断出 iSignal 值不等于 0，所以为假值，会执行 else 后的语句块。else 后的语句执行是我们不希望发生的，如图 6.8 所示。6.2.3 节将会提供解决这个问题的方法。

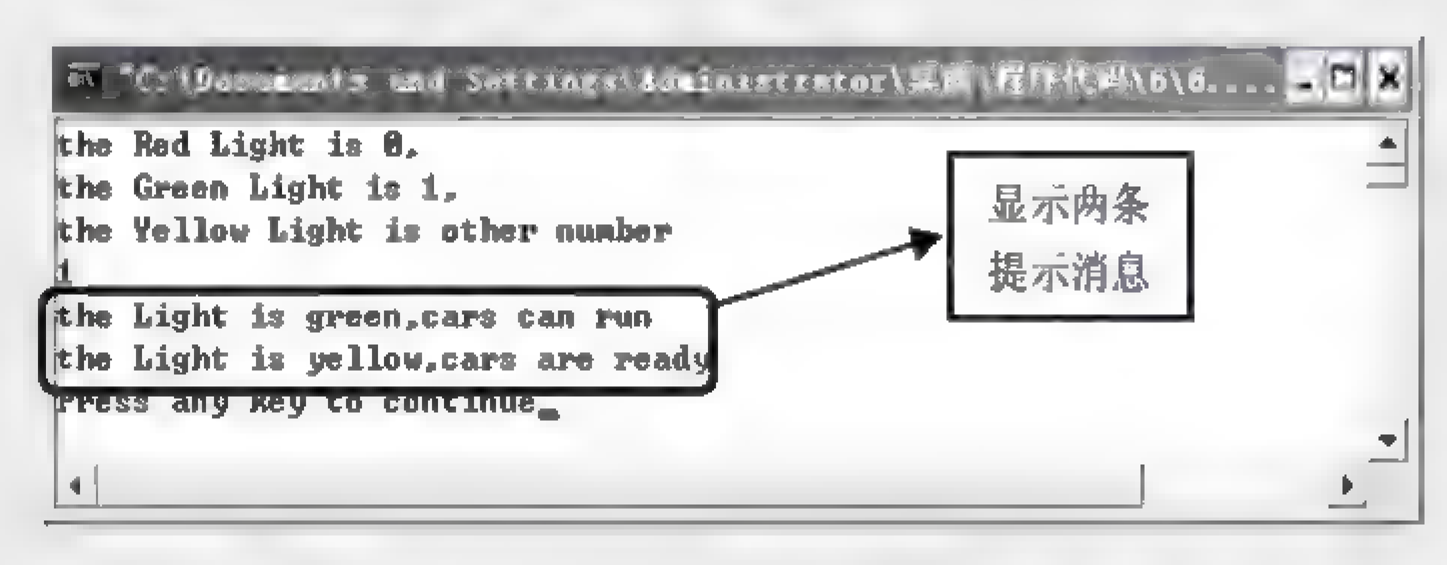


图 6.8　使用 if...else 语句模拟信号灯时可能出现的错误

6.2.3　else if 语句形式

利用 if 和 else 关键字的组合可以实现 else if 语句，这是对一系列互斥的条件进行检验，其一般形式如下：

```
if(表达式 1) 语句 1
else if(表达式 2) 语句 2
else if(表达式 3) 语句 3
    ...
else if(表达式 m) 语句 m
else 语句 n
```

else if 语句的执行流程图如图 6.9 所示。

根据流程图可知，首先对 if 语句中的表达式 1 进行判断，如果结果为真值，则执行后面跟着的语句 1，然后跳过 else if 语句和 else 语句，如果结果为假，那么判断 else if 语句中的表达式 2，如果表达式 2 为真值，那么执行语句 2 而不会执行后面 else if 的判断或者 else 语句。当所有的判断都不成立，也就是都为假值时，执行 else 后的语句块。例如：

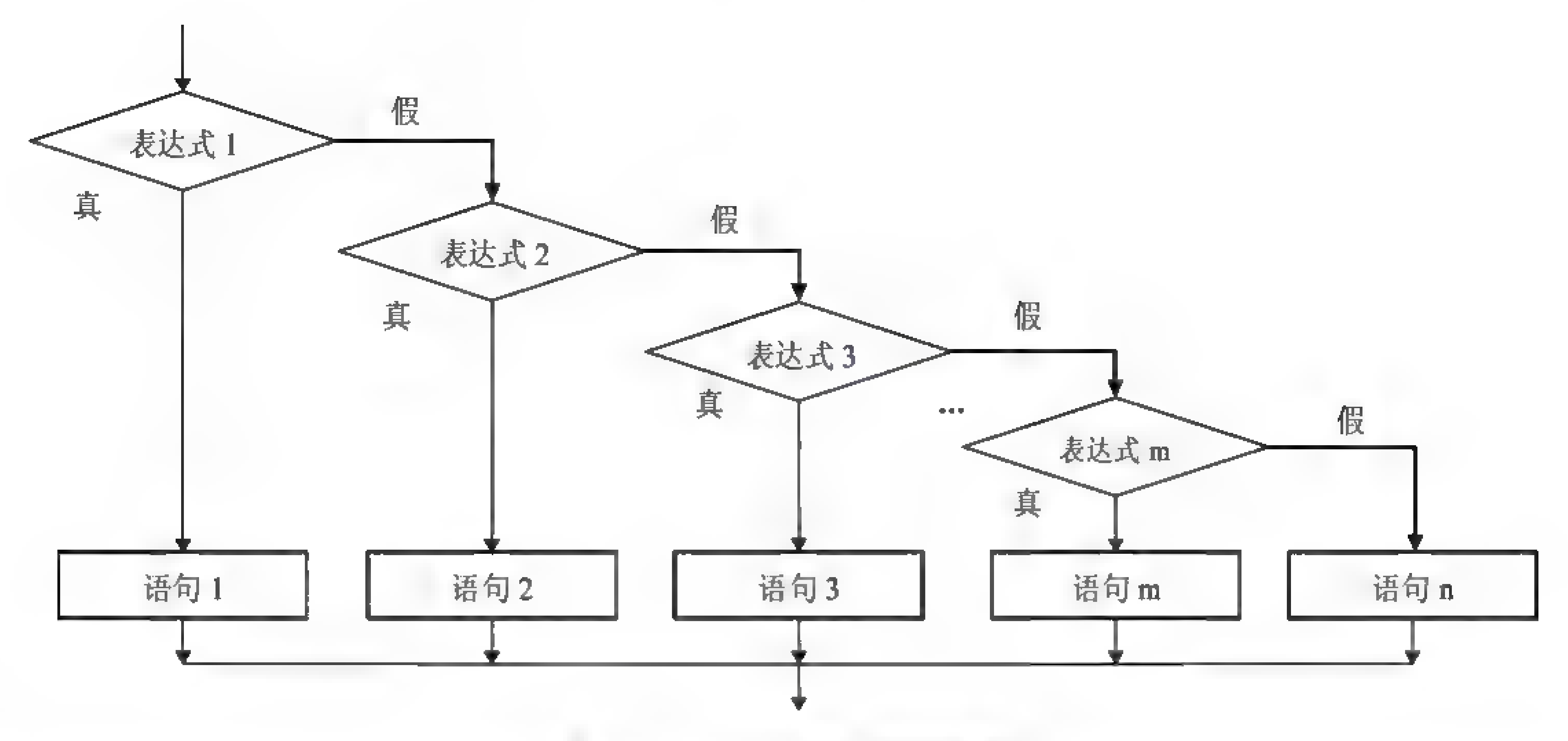


图 6.9　else if 语句的执行流程图

```
if(iSelection==1)
    {...}
else if(iSelection==2)
    {...}
else if(iSelection==3)
    {...}
else
    {...}
```

上述代码的含义是：使用 if 语句判断变量 iSelection 的值是否为 1，如果为 1，则执行后面语句块中的内容，然后跳过后面的 else if 判断和 else 语句的执行；如果 iSelection 的值不为 1，那么 else if 判断 iSelection 的值是否为 2，如果值为 2，则条件为真，执行后面紧跟着的语句块，执行完后跳过后面 else if 和 else 的操作；如果 iSelection 的值也不为 2，那么接下来的 else if 语句判断 iSelection 是否等于数值 3，如果等于，则执行后面语句块中的内容，否则执行 else 语句块中的内容。也就是说，当前面所有的判断都不成立（为假值）时，才执行 else 语句块中的内容。

【例 6.6】　使用 else if 语句编写屏幕菜单程序。（**实例位置：资源包\TM\sl\6\6**）

在本实例中，既然要对菜单进行选择，那么首先要显示菜单。利用格式输出函数将菜单中所需要的信息进行输出。

```
#include<stdio.h>

int main()
{
    int iSelection;                              /*定义变量，表示菜单的选项*/

    printf("---Menu---\n");                      /*输出屏幕的菜单*/
    printf("1 = Load\n");
    printf("2 = Save\n");
    printf("3 = Open\n");
    printf("other = Quit\n");
```

```
    printf("enter selection\n");                          /*提示信息*/
    scanf("%d",&iSelection);                              /*用户输入选项*/

    if(iSelection==1)                                     /*选项为 1*/
    {
        printf("Processing Load\n");
    }
    else if(iSelection==2)                                /*选项为 2*/
    {
        printf("Processing Save\n");
    }
    else if(iSelection==3)                                /*选项为 3*/
    {
        printf("Processing Open\n");
    }
    else                                                  /*选项为其他数值时*/
    {
        printf("Processing Quit\n");
    }
    return 0;
}
```

（1）程序中使用 printf 函数将可以进行选择的菜单显示输出，然后显示一条信息提示用户进行输入，选择一个菜单项进行操作。

（2）这里假设输入的数字为 3，变量 iSelection 将输入的数值保存，用来执行后续判断。

（3）再判断 iSelection 的位置，可以看到使用 if 语句判断 iSelection 是否等于 1，使用 else if 语句判断 iSelection 等于 2 和等于 3 的情况，如果都不满足，则会执行 else 处的语句。因为 iSelection 的值为 3，所以 iSelection==3 关系表达式为真，执行相应 else if 处的语句块，输出提示信息。

运行程序，显示效果如图 6.10 所示。



图 6.10　使用 else if 语句编写屏幕菜单程序

例 6.5 中使用 if...else 语句模拟信号灯时，连续使用两次 if 语句，当第一个 if 语句满足条件时会出现问题，因为 else 语句也会被执行。现在使用 else if 语句再一次修改此程序，使其功能完善。

【例 6.7】　使用 else if 语句正确修改信号灯程序。（**实例位置：资源包\TM\sl\6\7**）

```
#include<stdio.h>

int main()
```

```
{
    int iSignal;                                    /*定义变量表示信号灯的状态*/
    printf("the Red Light is 0,\nthe Green Light is 1,\nthe Yellow Light is other number\n");    /*输出提示信息*/
    scanf("%d",&iSignal);                           /*输入 iSignal 变量*/

    if(iSignal==1)                                  /*当信号灯为绿灯时*/
    {
        printf("the Light is green,cars can run\n");    /*判断结果为真时输出*/
    }
    else if(iSignal==0)                             /*当信号灯为红灯时*/
    {
        printf("the Light is red,cars can't run\n");    /*判断结果为真时输出*/
    }
    else                                            /*当信号灯为黄灯时*/
    {
        printf("the Light is yellow,cars are ready\n");
    }
    return 0;
}
```

在原来的程序中，只是将原来第二个 if 判断改成了 else if 判断。这样当输入“1”时程序就可以正常运行了。

通过对两个程序结果的比较可以发现，连续使用 if 判断条件这种方式中，每个条件的判断都是分开、独立的。而使用 if 和 else if 判断条件，所有的判断可以看成是一个整体，如果其中一个为真，那么下面 else if 中的判断即使有符合的，也会被跳过，不会执行。

运行程序，显示效果如图 6.11 所示。

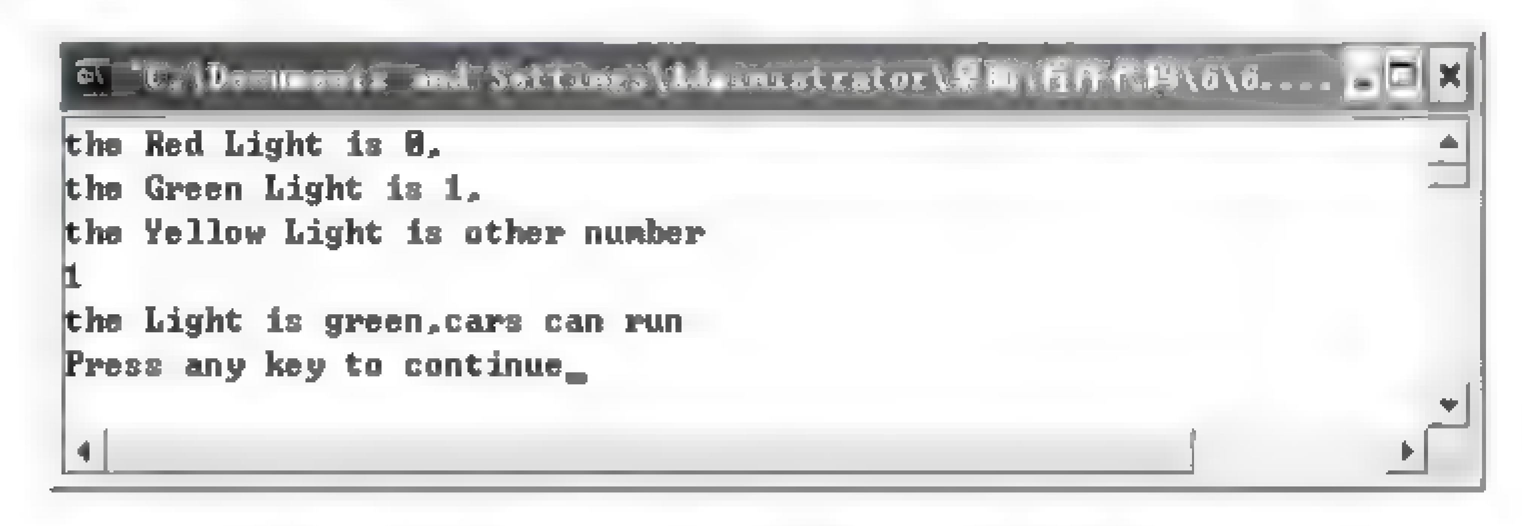


图 6.11　使用 else if 语句正确修改信号灯程序



6.3　if 的嵌套形式

if 语句中又包含一个或多个 if 语句，此种情况称为 if 语句的嵌套。一般形式如下：

```
if(表达式 1)
    if(表达式 2)    语句块 1
    else  语句块 2
else
    if(表达式 3)    语句块 3
    else  语句块 4
```

使用 if 语句的嵌套形式，可对判断的条件进行细化，然后进行相应的操作。

这就好比人们在生活中，每天早上醒来的时候通常会想一下今天是星期几，如果是周末就是休息日，如果不是周末就要上班。同时，休息日可以是星期六，也可以是星期日，星期六我们可以和朋友去逛街，星期日我们可以陪家人在家。

如何用 if 语句来实现上述比方呢？这里需要用到 if 嵌套语句。if 语句判断表达式 1，就像判断今天是星期几，假设判断结果为真，则用 if 语句判断表达式 2，这就好像判断出今天是休息日，然后去判断今天是不是星期六。如果 if 语句判断表达式 2 为真，那么执行语句块 1 中的内容；如果不为真，那么执行语句块 2 中的内容。例如，如果为星期六，就陪朋友逛街；如果为星期日，就陪家人在家。外面的 else 语句表示不为休息日时的相应操作。代码如下：

```
if(iDay>Friday)                          /*判断为休息日的情况*/
{
    if(iDay==Saturday)                   /*判断为星期六时的操作*/
    {}
    else                                 /*为星期日时的操作*/
    {}
}
else                                     /*不为休息日的情况*/
{
    if(iDay==Monday)                     /*判断为星期一时的操作*/
    {}
    else
    {}
}
```

上面的代码表示了整个 if 语句嵌套的操作过程，首先判断为休息日的情况，然后根据判断的结果选择相应的具体判断或者操作。过程如上述对 if 语句判断的描述。

注意

在使用 if 语句嵌套时，应注意 if 与 else 的配对情况。else 总是与其上面的最近的未配对的 if 进行配对。

前面曾经介绍过，使用 if 语句，如果只有一条语句，则可以不用大括号。修改一下上面的代码，让其先判断是否为工作日，然后在工作日中只判断星期一的情况。例如：

```
if(iDay<Friday)                          /*判断为休息日的情况*/
    if(iDay==Monday)                     /*判断为星期一时的操作*/
    {}
else
    if(iDay==Saturday)                   /*判断为星期六时的操作*/
    {}
    else
    {}
```

内嵌 if 语句块

原本这段代码的作用是先判断是否为工作日，是工作日，则判断是否为星期一；不是工作日，则

判断是否是星期六，否则就是星期日。但是因为 else 总是与其上面最近的未配对的 if 进行配对，所以 else 与第二个 if 语句配对，形成内嵌 if 语句块，这样就无法满足设计的要求。如果为 if 语句后的语句块加上大括号，就可避免出现这种情况了。因此，建议大家在条件判断中即使只有一条语句时也要使用大括号。

【例 6.8】 使用 if 嵌套语句选择日程安排。（**实例位置：资源包\TM\sl\6\8**）

在本实例中，使用 if 嵌套语句对输入的数据逐步进行判断，最终选择执行相应的操作。

```
#include<stdio.h>

int main()
{
	int iDay=0;                                          /*定义变量表示输入的星期*/
	/*定义变量代表一周中的每一天*/
	int Monday=1,Tuesday=2,Wednesday=3,Thursday=4,
		Friday=5,Saturday=6,Sunday=7;

	printf("enter a day of week to get course:\n");      /*提示信息*/
	scanf("%d",&iDay);                                   /*输入星期*/

	if(iDay>Friday)                                      /*休息日的情况*/
	{
		if(iDay==Saturday)                               /*为星期六时*/
		{
			printf("Go shopping with friends\n");
		}
		else                                             /*为星期日时*/
		{
			printf("At home with families\n");
		}
	}
	else                                                 /*工作日的情况*/
	{
		if(iDay==Monday)                                 /*为星期一时*/
		{
			printf("Have a meeting in the company\n");
		}
		else                                             /*为其他星期时*/
		{
			printf("Working with partner\n");
		}
	}
	return 0;
}
```

（1）在程序中定义变量 iDay 用来保存后面用户输入的数值，而其他变量表示一周中的每一天。

（2）在运行时，假设输入“6”，代表选择星期六。if 语句判断表达式 iDay>Friday，如果成立，则表示输入的是休息日，否则执行 else 表示工作日的部分。如果判断为真，则再利用 if 语句判断 iDay 是

否等于 Saturday 变量的值，如果等于，则表示为星期六，那么执行后面的语句，输出信息表示星期六和朋友去逛街。else 语句表示的是星期日，输出信息表示陪家人在家。

（3）因为 iDay 保存的数值为 6，大于 Friday，并且 iDay 等于 Saturday 变量的值，所以输出信息表示星期六要和朋友去逛街。

运行程序，显示效果如图 6.12 所示。



图 6.12　使用 if 嵌套语句选择日程安排



6.4　条件运算符

在使用 if 语句时，可以通过判断表达式为“真”或“假”，从而执行相应的表达式。例如：

```
if(a>b)
    {max=a;}
else
    {max=b;}
```

上面的代码可以用条件运算符“?:”来进行简化，例如：

```
max=(a>b)?a:b;
```

条件运算符可对一个表达式的值的真假情况进行检验，然后根据检验结果返回另外两个表达式中的一个。条件运算符的一般形式如下：

```
表达式 1?表达式 2:表达式 3;
```

在运算中，首先对第一个表达式的值进行检验。如果值为真，则返回第二个表达式的结果值；如果值为假，则返回第 3 个表达式的结果值。例如，上面使用条件运算符的代码，首先判断表达式 a>b 是否成立，成立则说明结果为真，否则为假。当为真时，将 a 的值赋给 max 变量；如果为假，则将 b 的值赋给 max 变量。

【例 6.9】　使用条件运算符计算欠款金额。（**实例位置：资源包\TM\sl\6\9**）

在本实例中，还欠款时，还钱的时间如果过期，则会在欠款的金额上增加 10%的罚款。其中使用条件运算符进行判断选择。

```
#include<stdio.h>

int main()
{
    float fDues;                                    /*定义变量表示欠款数*/
```

```
        float fAmount;                              /*表示要还的总欠款数*/
        int iOntime;                                /*表示是否按时归还*/
        char cChar;                                 /*用来接收用户输入的字符*/

        printf("Enter dues amount:\n");             /*显示信息，提示输入欠款金额*/
        scanf("%f",&fDues);                         /*用户输入*/
        printf("On Time? (y/n)\n");                 /*显示信息，提示是否按时还款*/
        getchar();                                  /*得到回车符*/
        cChar=getchar();                            /*得到输入的字符*/
        iOntime=(cChar=='y')?1:0;                   /*使用条件运算符根据字符选择进行选择操作*/
        fAmount=iOntime?fDues:(fDues*1.1);          /*使用条件运算符根据 iOntime 值的真假进行选择操作*/
        printf("the Amount is:%.2f\n",fAmount);     /*将计算应还的总欠款数输出*/
        return 0;
}
```

（1）定义变量 fDues 表示欠款的金额，fAmount 表示应该还款的金额，iOntime 的值表示有没有按时还款，cChar 用字符表示有没有按时还款。

（2）提醒用户输入数据。假设用户输入的欠款金额为 100，然后提示有没有按时还款。用户输入“y”表示按时还款，“n”表示没有按时还款。

（3）假设用户输入“n”，表示没有按时还款，接下来使用条件运算符判断表达式 cChar=='y'是否成立。成立为真时，将“?”号后的值 1 赋给 iOntime 变量；不成立为假时，将 0 赋给 iOntime 变量。因为 cChar=='y'的表达式不成立，所以 iOntime 的值为 0。

（4）使用条件运算符对 iOntime 的值进行判断。如果 iOntime 为真，则说明按时还款，还款额为原来的欠款，返回 fDues 值给 fAmount 变量。若 iOntime 值为假，则说明没有按时还款，要加上 10%的罚金，返回表达式 fDues*1.1 的值给 fAmount 变量。这里 iOntime 为 0，因此 fAmount 值为 fDues*1.1 的结果。

运行程序，显示效果如图 6.13 所示。



图 6.13 使用条件运算符计算欠款金额



6.5 switch 语句

从前文的介绍可知，if 语句只有两个分支可供选择，而在实际问题中常需要用到多分支的选择。当然，使用嵌套的 if 语句也可以实现多分支的选择，但是如果分支较多，就会使得嵌套的 if 语句层数较多，程序冗余，并且可读性不好。C 语言中可以使用 switch 语句直接处理多分支选择的情况，将程

序的代码可读性提高。

6.5.1　switch 语句的基本形式

switch 语句是多分支选择语句。例如，如果只需要检验某一个整型变量的可能取值，那么可以用更简便的 switch 语句。switch 语句的一般形式如下：

```
switch(表达式)
{
    case 情况 1:
        语句块 1;
    case 情况 2:
        语句块 2;
    …
    case 情况 n:
        语句块 n;
    default:
        默认情况语句块;
}
```

switch 语句的执行流程图如图 6.14 所示。

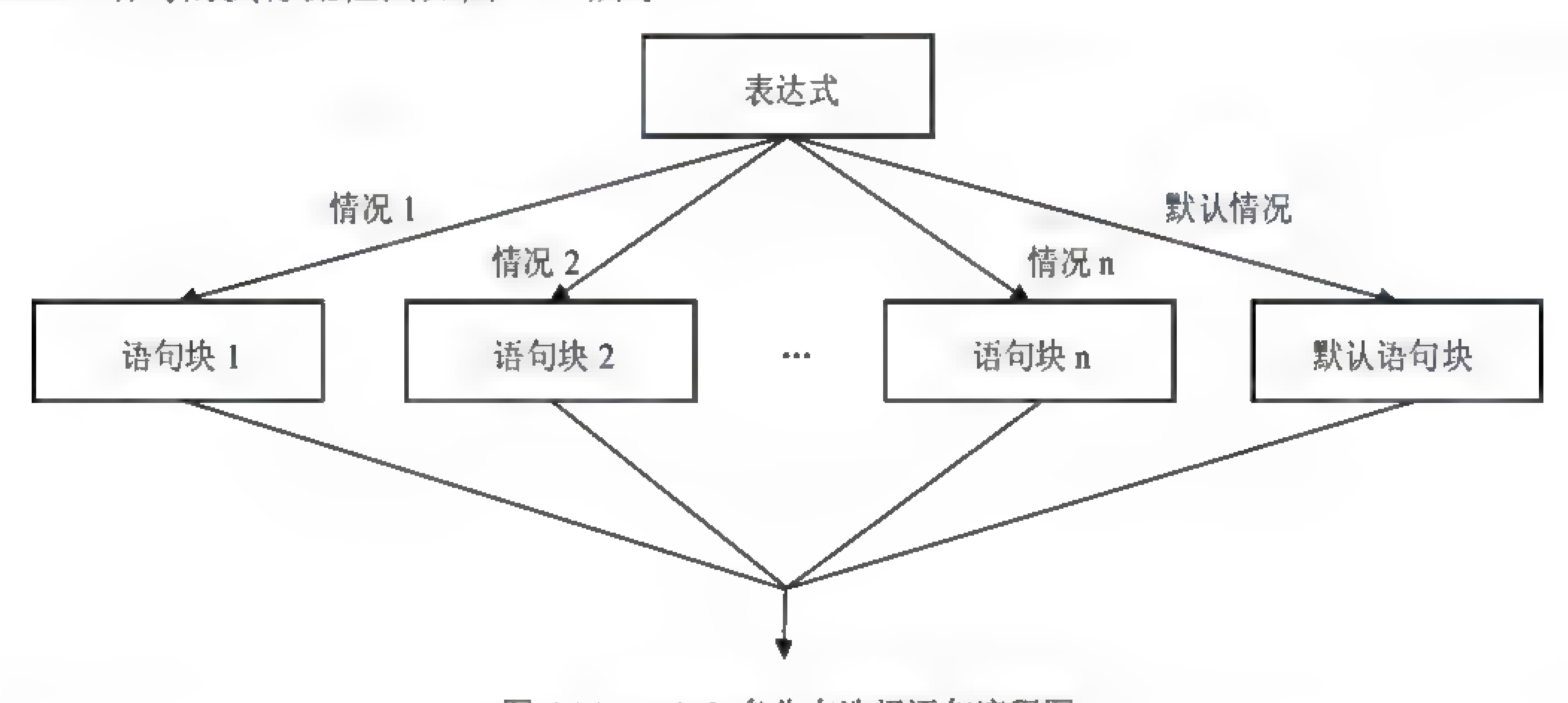


图 6.14　switch 多分支选择语句流程图

通过上面的流程图可知，switch 后面括号中的表达式就是要进行判断的条件。在 switch 的语句块中，使用 case 关键字表示检验条件符合的各种情况，其后的语句是相应的操作。其中还有一个 default 关键字，作用是如果没有符合条件的情况，那么执行 default 后的默认情况语句。


说明

switch 语句检验的条件必须是一个整型表达式，这意味着其中也可以包含运算符和函数调用。而 case 语句检验的值必须是整型常量，即常量表达式或者常量运算。

通过如下代码再来分析一下 switch 语句的使用方法：

```
switch(selection)
    {
    case 1:
        printf("Processing Receivables\n");
        break;
    case 2:
        printf("Processing Payables\n");
        break;
    case 3:
        printf("Quitting\n");
        break;
    default:
        printf("Error\n");
        break;
    }
```

其中 switch 判断 selection 变量的值，利用 case 语句检验 selection 值的不同情况。假设 selection 的值为 2，那么执行 case 为 2 时的情况，执行后跳出 switch 语句。如果 selection 的值不是 case 中所检验列出的情况，那么执行 default 中的语句。在每一个 case 或 default 语句后都有一个 break 关键字。break 语句用于跳出 switch 结构，不再执行 switch 下面的代码。

注意

在使用 switch 语句时，如果没有一个 case 语句后面的值能匹配 switch 语句的条件，就执行 default 语句后面的代码。其中，任意两个 case 语句都不能使用相同的常量值；每个 switch 结构只能有一个 default 语句，而且 default 可以省略。

【例 6.10】 使用 switch 语句输出分数段。（**实例位置：资源包\TM\sl\6\10**）

在本实例中，要求按照考试成绩的等级输出分数段，其中要使用 switch 语句来判断分数的情况。

```
#include<stdio.h>

int main()
{
    char cGrade;                                    /*定义变量表示分数的级别*/
    printf("please enter your grade\n");            /*提示信息*/
    scanf("%c",&cGrade);                            /*输入分数的级别*/
    printf("Grade is about:");                      /*提示信息*/
    switch(cGrade)                                  /*switch 语句判断*/
    {
    case 'A':                                       /*分数级别为 A 的情况*/
        printf("90~100\n");                         /*输出分数段*/
        break;                                      /*跳出*/
    case 'B':                                       /*分数级别为 B 的情况*/
        printf("80~89\n");                          /*输出分数段*/
        break;                                      /*跳出*/
    case 'C':                                       /*分数级别为 C 的情况*/
        printf("70~79\n");                          /*输出分数段*/
        break;                                      /*跳出*/
```

```
    case 'D':                                    /*分数级别为 D 的情况*/
        printf("60~69\n");                       /*输出分数段*/
        break;                                   /*跳出*/
    case 'F':                                    /*分数级别为 F 的情况*/
        printf("<60\n");                         /*输出分数段*/
        break;                                   /*跳出*/
    default:                                     /*默认情况*/
        printf("You enter the char is wrong!\n"); /*提示错误*/
        break;                                   /*跳出*/
    }
    return 0;
}
```

（1）在程序代码中，定义变量 cGrade 用来保存用户输入的成绩并判定级别。

（2）使用 switch 语句判断字符变量 cGrade，其中使用 case 关键字检验可能出现的级别情况，并且在每一个 case 语句的最后都会有 break 进行跳出。如果没有符合的情况，则会执行 default 默认语句。

注意

在 case 语句表示的条件后有一个冒号“:”，在编写程序时不要忘记。

（3）在程序中，假设用户输入字符为 B，在 case 检验中有为 B 的情况，那么执行该 case 后的语句块，将分数段进行输出。

运行程序，显示效果如图 6.15 所示。



图 6.15　使用 switch 语句输出分数段

在使用 switch 语句时，每一个 case 情况中都要使用 break 语句。如果不使用 break 语句，会出现什么情况呢？先来看一下 break 的作用：break 使得执行完 case 语句后跳出 switch 语句。进行一下猜测，如果没有 break 语句说明，程序可能会将后面的内容都执行。为了验证猜测是否正确，将上面程序中的 break 注释掉，还是输入字符“B”，运行程序，显示结果如图 6.16 所示。

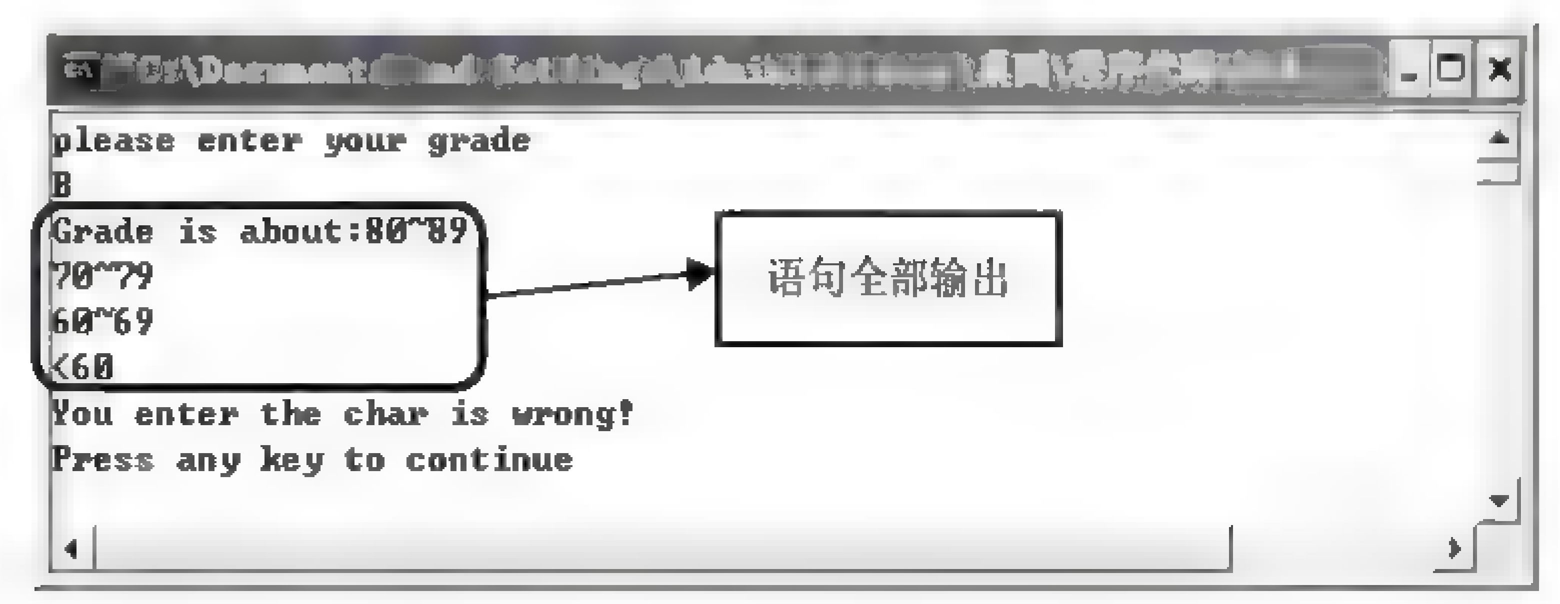


图 6.16　不添加 break 的情况

从运行结果可以看出，当去掉 break 语句后，会将 case 检验相符情况后的所有语句进行输出。因

此，break 语句在 case 语句中是不能缺少的。

【例 6.11】 修改日程安排程序。（**实例位置：资源包\TM\sl\6\11**）

在例 6.8 中，使用嵌套的 if 语句形式编写了日程安排程序，这里要求使用 switch 语句对程序进行修改。

```
#include<stdio.h>

int main()
{
    int iDay=0;                                        /*定义变量表示输入的星期*/

    printf("enter a day of week to get course:\n");    /*提示信息*/
    scanf("%d",&iDay);                                 /*输入星期*/

    switch(iDay)
    {
    case 1:                                            /*iDay 的值为 1 时*/
        printf("Have a meeting in the company\n");
        break;
    case 6:                                            /*iDay 的值为 6 时*/
        printf("Go shopping with friends\n");
        break;
    case 7:                                            /*iDay 的值为 7 时*/
        printf("At home with families\n");
        break;
    default:                                           /*iDay 的值为其他情况时*/
        printf("Working with partner\n");
        break;
    }
    return 0;
}
```

在程序中，使用 switch 语句将原来的 if 语句都去掉，使得程序的结构比较清晰，易于观察。在使用 case 进行检验时，不要忘记 case 检验的条件只能是常量或者常量表达式，而不能对变量进行检验。

运行程序，显示效果如图 6.17 所示。



图 6.17 修改日程安排程序

6.5.2 多路开关模式的 switch 语句

在例 6.11 中，将 break 去掉之后，会将符合检验条件后的所有语句都输出。利用这个特点，可以设计多路开关模式的 switch 语句，其形式如下：

```
switch(表达式)
{
    case 1:
        语句 1
        break;
    case 2:
    case 3:
        语句 2
        break;
    …
    default:
        默认语句
        break;
}
```

可以看到，如果在 case 2 后不使用 break 语句，那么符合检验时与符合 case 3 检验时的效果是一样的。也就是说，使用多路开关模式使得多种检验条件使用一种解决方式。

【例 6.12】　使用多路开关模式编写日程安排程序。（**实例位置：资源包\TM\sl\6\12**）

在本实例中，要求将操作相同的检验结果使用多路开关的模式进行编写，当输入不正确的日期时，进行错误提示。

```
#include<stdio.h>

int main()
{
    int iDay=0;                                         /*定义变量表示输入的星期*/

    printf("enter a day of week to get course:\n");     /*提示信息*/
    scanf("%d",&iDay);                                  /*输入星期*/

    switch(iDay)
    {
    case 1:                                             /*iDay 的值为 1 时*/
        printf("Have a meeting in the company\n");
        break;
    /*多路开关模式*/
    case 2:
    case 3:
    case 4:
    case 5:
        printf("Working with partner\n");
        break;
    case 6:                                             /*iDay 的值为 6 时*/
        printf("Go shopping with friends\n");
        break;
    case 7:                                             /*iDay 的值为 7 时*/
        printf("At home with families\n");
        break;
    default:                                            /*iDay 的值错误时*/
```

```
            printf("error!!\n");
        }
        return 0;
}
```

在程序中使用多路开关模式，使得检测 iDay 的值为 2、3、4、5 这 4 种情况时，都会执行相同的结果，并且利用 default 语句将不符合的数字显示，提示信息表示输入错误。

运行程序，显示效果如图 6.18 所示。



图 6.18　使用多路开关模式编写日程安排程序



6.6　if…else 语句和 switch 语句的区别

if…else 语句和 switch 语句都用于根据不同的情况检验条件并做出相应的判断。那么 if…else 语句和 switch 语句有什么区别呢？下面从两者的语法和效率方面进行比较。

1．语法的比较

if 需要配合 else 关键字进行使用，switch 需要配合 case 关键字进行使用；if 语句是先对条件进行判断，而 switch 语句是后进行判断。

2．效率的比较

if…else 结构对少量的检验，判断速度比较快，但是随着检验的增长，会逐渐变慢，其中的默认情况是最慢的。使用 if…else 结构可以判断表达式，但同样存在随着深度的增加检验速度逐渐变慢的问题，并且也不容易进行后续的添加扩充。

switch 结构中，除了 default 默认情况下，对其他每一项 case 的检验速度都是相同的。default 默认情况比其他情况都快。

当需要判定的情况较少时，使用 if…else 结构比使用 switch 结构检验速度快。也就是说，如果分支在 3 个或者 4 个以下，用 if…else 结构比较好，否则应选择 switch 结构。

【例 6.13】　if…else 语句和 switch 语句的综合应用。（**实例位置：资源包\TM\sl\6\13**）

在本实例中，要求设计如下程序：输入一年中的月份，得到这个月所包含的天数。用户须判断数量的情况，根据需求选择使用 if…else 语句或 switch 语句。

```
#include<stdio.h>

int main()
```

```
{
    int iMonth=0,iDay=0;                                    /*定义变量*/
    printf("enter the month you want to know the days\n");  /*提示信息*/
    scanf("%d",&iMonth);                                    /*输入数据*/
    switch(iMonth)                                          /*检验变量*/
    {
    /*多路开关模式 switch 语句进行检验*/
    case 1:                                                 /*1 表示 1 月份*/
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        iDay=31;                                            /*iDay 赋值为 31*/
        break;                                              /*跳出 switch 结构*/
    case 4:
    case 6:
    case 9:
    case 11:
        iDay=30;                                            /*iDay 赋值为 30*/
        break;                                              /*跳出 switch 结构*/
    case 2:
        iDay=28;                                            /*iDay 赋值为 28*/
        break;                                              /*跳出 switch 结构*/
    default:                                                /*默认情况*/
        iDay=-1;                                            /*赋值为-1*/
        break;                                              /*跳出 switch 结构*/
    }

    if(iDay==-1)                                            /*使用 if 语句判断 iDay 的值*/
    {
        printf("there is a error with you enter\n");
    }
    else                                                    /*默认的情况*/
    {
        printf("2010.%d has %d days\n",iMonth,iDay);
    }
    return  0;
}
```

要判断一年中 12 个月份所包含的日期数，就要对 12 种不同的情况进行检验。由于检验数量比较多，所以使用 switch 结构判断月份比较合适，并且可以使用多路开关模式，使得程序更为简洁。其中 case 语句用来判断月份 iMonth 的情况，并且为 iDay 赋相应的值。default 默认处理为输入的月份不符合检验条件时，iDay 赋值为-1。

switch 检验完成之后，要输出得到的日期数。因为有可能日期为-1，也就是出现月份错误的情况。这时判断的情况只有两种，就是 iDay 是否为-1，因此检验的条件少，所以使用 if 语句更为方便。

运行程序，显示效果如图 6.19 所示。

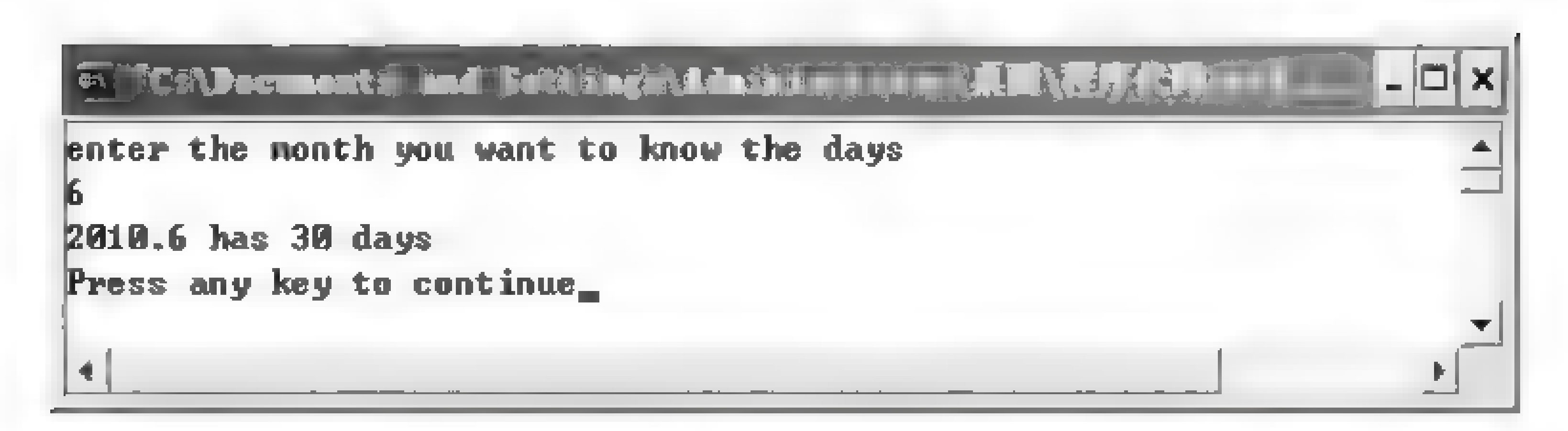


图 6.19 if...else 语句和 switch 语句的综合应用

6.7 小　　结

本章介绍了选择结构的程序设计方式，包括 if 语句和 switch 语句。同时对 if...else 语句和 else if 语句的形式也进行了介绍，为选择结构程序提供了更多的控制方式。然后介绍了 switch 语句，switch 语句用在检验条件较多的情况，此时使用 if 语句进行嵌套也是可以实现的，不过其程序的可读性会降低。最后通过两种选择语句的比较来进行区分。

掌握选择结构的程序设计方法是必要的，这是程序设计的重点。

6.8 实践与练习

1．利用选择结构设计一个程序，使其能计算函数：

$$\begin{cases} y=x & (x<1) \\ y=2x-1 & (1\leqslant x<10) \\ y=3x-11 & (x\geqslant 10) \end{cases}$$

当输入 x 值时，计算显示 y 值。**（答案位置：资源包\TM\sl\6\14）**

2．设计一个程序，要求通过键盘输入 3 个任意的整数，并输出其中最大的数。**（答案位置：资源包\TM\sl\6\15）**

第 7 章

循环控制

（视频讲解：43 分钟）

日常生活中总会有许多简单而重复的工作，为完成这些重复性工作，需要花费很多时间。而编写程序的目的就是使工作变得简单，使用计算机来处理这些重复的工作是最好不过的了。

本章致力于使读者了解循环语句的特点，分别介绍了 while 语句结构、do...while 语句结构和 for 语句结构 3 种循环结构，并且对这 3 种循环结构进行区分讲解，帮助读者掌握转移语句的相关内容。

通过阅读本章，您可以：

- 了解循环语句的概念
- 掌握 while 循环语句的使用方式
- 掌握 do...while 循环语句的使用方式
- 掌握 for 循环语句的使用方式
- 区分 3 种循环语句的不同特点和嵌套使用方式
- 能够使用转移语句控制程序的流程



7.1 循环语句

从第 6 章的介绍了解到，程序在运行时可以通过判断、检验条件做出选择。此处，程序还必须能够重复，也就是反复执行一段指令，直到满足某个条件为止。例如，要计算一个公司的消费总额，就要将所有的消费加起来。

这种重复的过程就称为循环。C 语言中有 3 种循环语句，即 while、do...while 和 for 循环语句。循环结构是结构化程序设计的基本结构之一，因此熟练掌握循环结构是程序设计的基本要求。



7.2 while 语句

使用 while 语句可以执行循环结构，其一般形式如下：

```
while(表达式)语句
```

while 语句的执行流程图如图 7.1 所示。

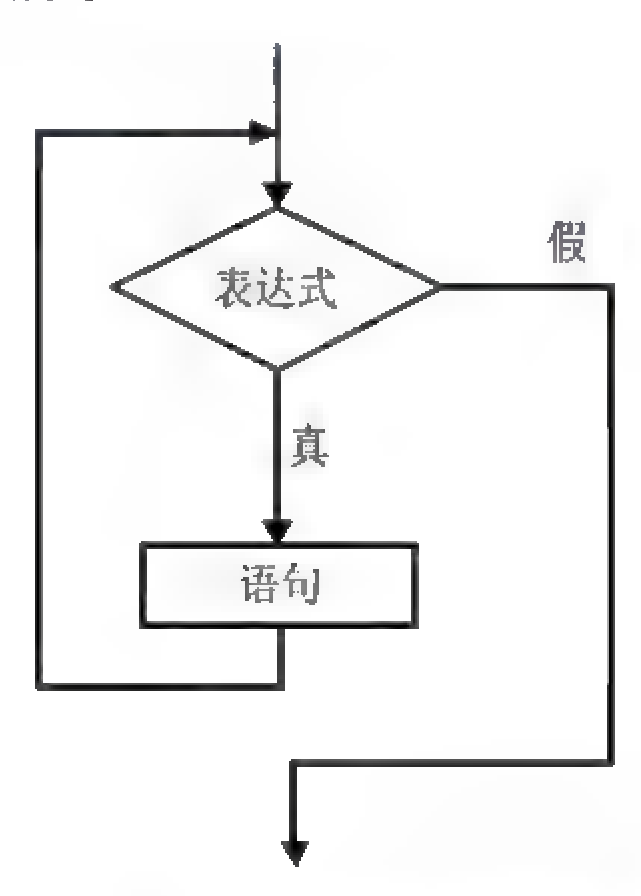


图 7.1　while 语句的执行流程图

while 语句首先检验一个条件，也就是括号中的表达式。当条件为真时，就执行紧跟其后的语句或者语句块。每执行一遍循环，程序都将回到 while 语句处，重新检验条件是否满足。如果一开始条件就不满足，则跳过循环体中的语句，直接执行后面的程序代码。如果第一次检验时条件满足，那么在第一次或其后的循环过程中，必须得有使条件为假的操作，否则循环将无法终止。



无法终止的循环常被称为死循环或者无限循环。

例如下面的代码：

```
while(iSum<100)
{
    iSum+=1;
}
```

在这段代码中，while 语句首先判断 iSum 变量是否小于常量 100，如果小于 100，为真，那么执行紧跟其后的语句块；如果不小于 100，为假，那么跳过语句块中的内容直接执行下面的程序代码。在语句块中，可以看到对其中的变量进行加 1 的运算，这里的加 1 运算就是循环结构中使条件为假的操作，也就是使得 iSum 不小于 100，否则程序会一直循环下去。

【例 7.1】　计算 1 累加到 100 的结果。（**实例位置：资源包\TM\sl\7\1**）

本实例计算从数字 1～100 所有数字的总和，使用循环语句可以将 1～100 的数字进行逐次加运算，直到 while 判断的条件不满足为止。

```
#include<stdio.h>

int main()
{
    int iSum=0;                              /*定义变量，表示计算总和*/
    int iNumber=1;                           /*表示每一个数字*/

    while(iNumber<=100)                      /*使用 while 循环*/
    {
        iSum=iSum+iNumber;                   /*进行累加*/
        iNumber++;                           /*增加数字*/
    }
    printf("the result is：%d\n",iSum);      /*将结果输出*/
    return 0;
}
```

（1）在程序代码中，因为要计算 1～100 的累加结果，所以要定义两个变量，iSum 表示计算的结果，iNumber 表示 1～100 的数字。为 iSum 赋值为 0，为 iNumber 赋值为 1。

（2）使用 while 语句判断 iNumber 是否小于等于 100，如果条件为真，则执行其后语句块中的内容；如果条件为假，则跳过语句块执行后面的内容。初始 iNumber 的值为 1，判断的条件为真，因此执行语句块。

（3）在语句块中，总和 iSum 等于先前计算的总和结果加上现在 iNumber 表示的数字，完成累加操作。iNumber++表示自身加 1 操作，语句块执行结束，while 再次判断新的 iNumber 值。也就是说，“iNumber++;”语句可以使得循环停止。

（4）当 iNumber 大于 100 时，循环操作结束，将结果 iSum 输出。

运行程序，显示效果如图 7.2 所示。

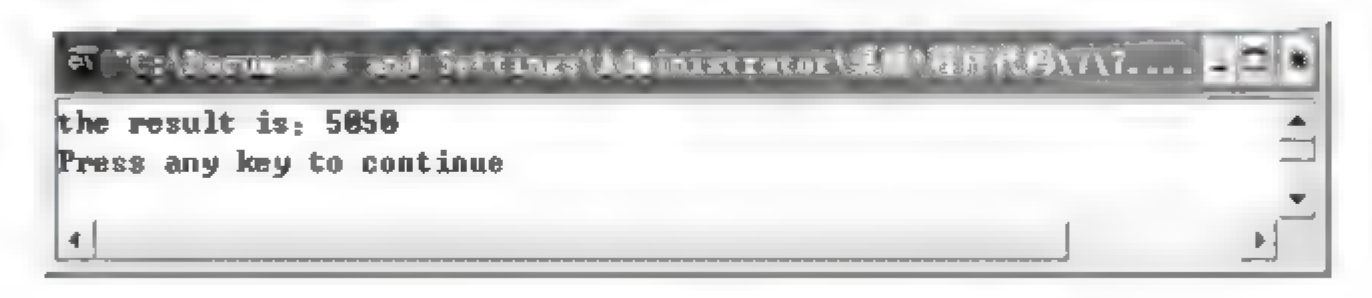


图 7.2　计算 1 累加到 100 的结果

【例 7.2】 使用 while 循环语句为用户提供菜单显示。（**实例位置：资源包\TM\sl\7\2**）

在使用程序时，根据程序的功能会有许多选项，为了使用户可以方便地观察到菜单的选项，要将其菜单进行输出。在本实例中，利用 while 循环语句将菜单进行循环输出，这样可以使用户更为清楚地知道每一个选项所对应的操作。

```
#include<stdio.h>

int main()
{
    int iSelect=1;                                          /*定义变量，表示菜单的选项*/

    while(iSelect!=0)                                       /*检验条件，循环显示菜单*/
    {
        /*显示菜单内容*/
        printf("--------Menu--------\n");
        printf("----Sell---------1\n");
        printf("----Buy----------2\n");
        printf("----ShowProduct---3\n");
        printf("----Out----------0\n");

        scanf("%d",&iSelect);                               /*输入菜单的选项*/
        switch(iSelect)                                     /*使用 switch 语句，检验条件进行相应处理*/
        {
        case 1:                                             /*选择第一项菜单的情况*/
            printf("you are buying something into store\n");
            break;
        case 2:                                             /*选择第二项菜单的情况*/
            printf("you are selling to consumer\n");
            break;
        case 3:                                             /*选择第三项菜单的情况*/
            printf("checking the store\n");
            break;
        case 0:                                             /*选择退出项菜单的情况*/
            printf("the Program is out\n");
            break;
        default:                                            /*默认处理*/
            printf("You put a wrong selection\n");
            break;
        }
    }
    return 0;
}
```

（1）在程序代码中，定义的变量 iSelect 用来保存后面的选项。使用 while 语句检验 iSelect 变量，iSelect!=0 表示如果 iSelect 不等于 0，则说明条件为真。为真时，执行其后语句块中的内容；为假时，执行后面的代码，return 0 程序结束。

（2）因为设定 iSelect 变量的值为 1，所以 while 语句刚进行检验时为真，执行其中的语句块。在

语句块中首先显示菜单，将每一项操作都进行说明。

（3）使用 scanf 语句，用户将要进行选择的项目输入。然后使用 switch 语句判断变量，根据变量中保存的数据，检验出相对应的结果进行操作，其中每一个 case 中输出不同的提示信息表示不同的功能。default 为默认情况，当用户输入的选项为菜单所列选项以外时的操作。

（4）显示的菜单中有 4 项功能，其中的选项 0 为退出。输入“0”时，iSelect 保存 0 值。这样在执行完 case 为 0 的情况后，当 while 再检验 iSelect 的值时，判断的结果为假，则不执行循环操作，执行后面的代码后，程序结束。

运行程序，显示效果如图 7.3 所示。

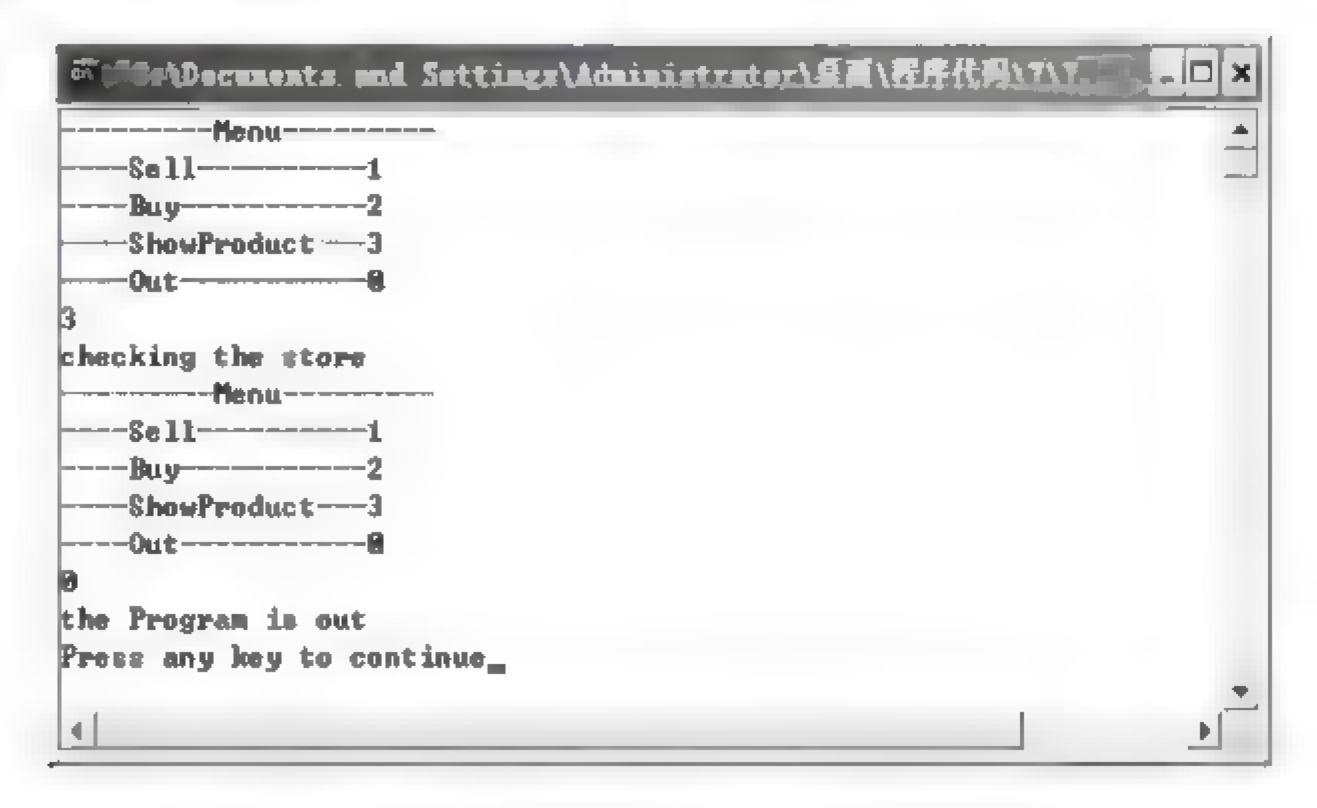


图 7.3　使用 while 循环语句为用户提供菜单显示



7.3　do...while 语句

在有些情况下，不论条件是否满足，循环过程必须至少执行一次，这时可以采用 do...while 语句。do...while 语句的特点就是先执行循环体语句的内容，然后判断循环条件是否成立。其一般形式如下：

```
do
    循环体语句
while(表达式);
```

do...while 语句的执行流程图如图 7.4 所示。

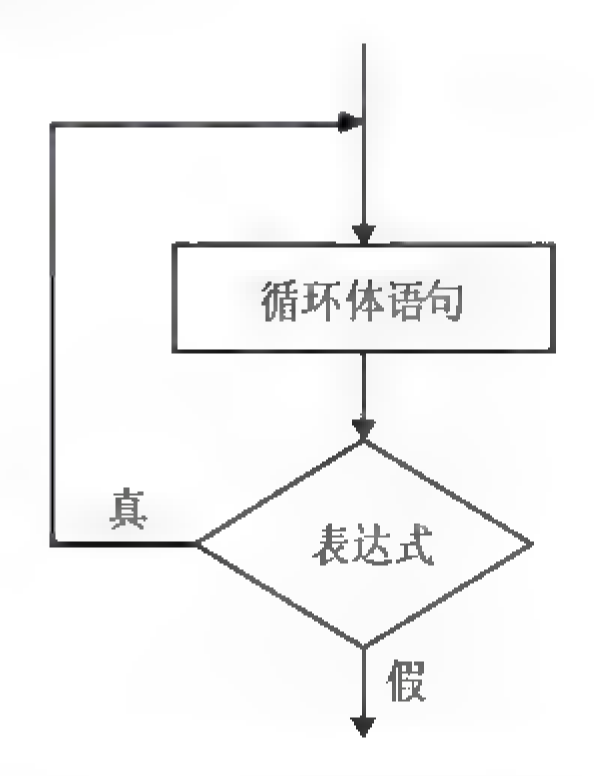


图 7.4　do...while 语句的执行流程图

do…while 语句是这样执行的，首先执行一次循环体语句中的内容，然后判断表达式，当表达式的值为真时，返回重新执行循环体语句。执行循环，直到表达式的判断为假时为止，此时循环结束。

说明

while 语句和 do…while 语句的区别在于：while 语句在每次循环之前检验条件，do…while 语句在每次循环之后检验条件。这也可以从两种循环结构的代码上看出来，while 结构中的 while 语句出现在循环体的前面，do…while 结构中的 while 语句出现在循环体的后面。

例如下面的代码：

```
do
{
    iNumber++;
}
while(iNumber<100);
```

在上面的代码中，首先执行 iNumber++的操作，也就是说，不管 iNumber 是否小于 100，都会执行一次循环体中的内容。然后判断 while 后括号中的内容，如果 iNumber 小于 100，则再次执行循环语句块中的内容；条件为假时，执行下面的程序操作。

注意

在使用 do…while 语句时，条件要放在 while 关键字后面的括号中，最后必须加上一个分号，这是许多初学者容易忘记的。

【例 7.3】 使用 do…while 语句计算 1～100 的累加结果。（**实例位置：资源包\TM\sl\7\3**）

在 7.2 节中，计算 1～100 所有数字的累加方法使用的是 while 语句，在本实例中使用 do…while 语句实现相同的功能。在程序运行过程中，虽然两者的结果是相同的，但是要了解其中操作的区别。

```
#include<stdio.h>

int main()
{
    int iNumber=1;                          /*定义变量，表示数字*/
    int iSum=0;                             /*表示计算的总和*/

    do
    {
        iSum=iSum+iNumber;                  /*计算累加的总和*/
        iNumber++;                          /*进行自身加 1*/
    }
    while(iNumber<=100);                    /*检验条件*/

    printf("the result is: %d\n",iSum);     /*输出计算结果*/
    return 0;
}
```

（1）在程序中，同样定义 iNumber 表示 1～100 的数字，而 iSum 表示计算的总和。

（2）do 关键字之后是循环语句，在语句块中进行累加操作，并对 iNumber 变量进行自加操作。语句块下方是 while 语句检验条件，如果检验为真，则继续执行上面的语句块操作；为假时，程序执行下面的代码。

（3）在循环操作完成之后，将结果输出。

运行程序，显示效果如图 7.5 所示。

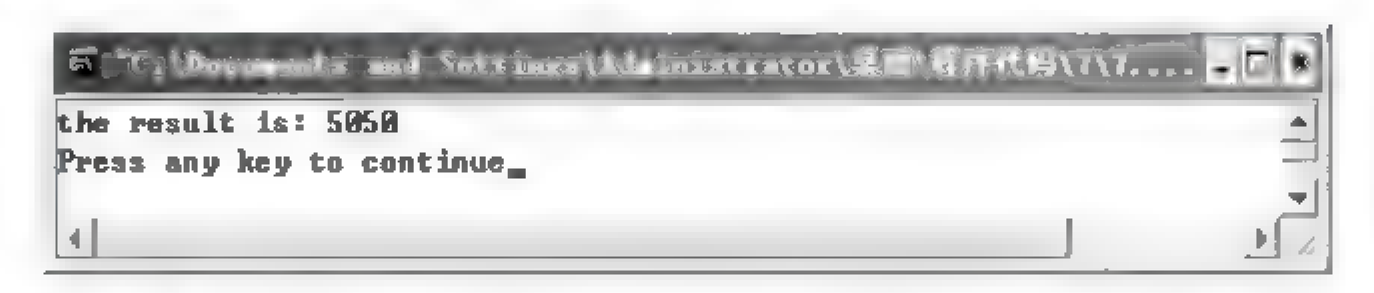


图 7.5 使用 do...while 语句计算 1～100 的累加结果

7.4 for 语句



C 语言中，使用 for 语句也可以控制一个循环，并且在每一次循环时修改循环变量。在循环语句中，for 语句的应用最为灵活，不仅可以用于循环次数已经确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况。下面将对 for 语句的循环结构进行详细的介绍。

7.4.1 for 语句使用

for 语句的一般形式如下：

```
for(表达式 1;表达式 2;表达式 3;)
```

每条 for 语句包含 3 个用分号隔开的表达式。这 3 个表达式用一对圆括号括起来，其后紧跟着循环语句或语句块。当执行到 for 语句时，程序首先计算第一个表达式的值，接着计算第二个表达式的值。如果第二个表达式的值为真，程序就执行循环体的内容，并计算第 3 个表达式；然后检验第二个表达式，执行循环；如此反复，直到第二个表达式的值为假，退出循环。

for 语句的执行流程如图 7.6 所示。

通过上面的流程图和对 for 语句的介绍，总结其执行过程如下：

（1）求解表达式 1。

（2）求解表达式 2，若其值为真，则执行 for 语句中的循环语句块，然后执行步骤（3）；若为假，则结束循环，转到步骤（5）。

（3）求解表达式 3。

（4）回到上面的步骤（2），继续执行。

（5）循环结束，执行 for 语句下面的一个语句。

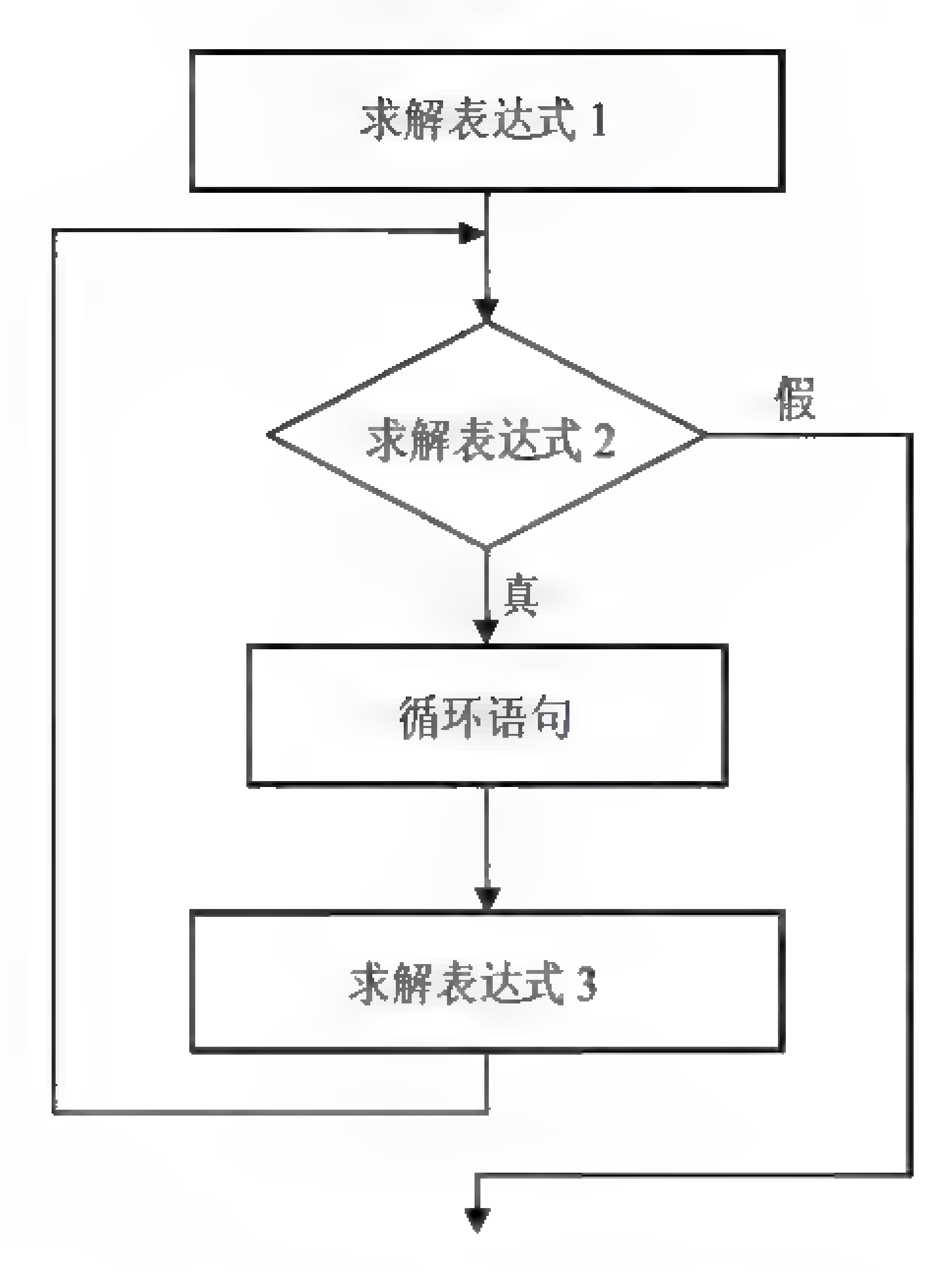


图 7.6　for 语句的执行流程图

其实 for 语句简单的应用形式如下：

```
for(循环变量赋初值;循环条件;循环变量) 语句块
```

例如实现一个循环操作：

```
for(i=1;i<100;i++)
{
    printf("the i is:%d",i);
}
```

在上面的代码中，表达式 1 是对循环变量 i 进行赋值操作，表达式 2 是判断循环条件是否为真。因为 i 的初值为 1，小于 100，所以执行语句块中的内容。表达式 3 用于在每次循环结束后，对循环变量进行自增操作，然后继续判断表达式 2 的状态。为真时，继续执行语句块；为假时，循环结束，执行后面的程序代码。

【例 7.4】　使用 for 语句显示随机数。(**实例位置：资源包\TM\sl\7\4**)

在本实例中，要求使用 for 循环语句显示 10 个随机数字。其中，产生随机数要用到 srand 和 rand 函数，这两个函数都包含在 stdio.h 头文件中。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int counter;                                            /*定义变量*/
    /*使用 for 语句，为变量赋值，执行循环*/
    for(counter=0;counter<10;counter++)
    {
        srand(counter+1);                                   /*设置随机发生数的种子*/
        printf("Random number %d is: %d\n",counter,rand());   /*产生随机发生数*/
```

```
    }
    return 0;
}
```

（1）在程序代码中，定义变量 counter。在 for 语句中先对 counter 进行赋值，然后判断 counter<10 的条件是否为真，再根据判断的结果选择是否执行循环语句。

（2）srand 和 rand 函数都包含在 stdio.h 头文件中，srand 函数的功能是设定一个随机发生数的种子，rand 函数是根据设定的随机发生数种子产生特定的随机数。

（3）在循环语句中使用 srand 函数设定 counter+1 为设定的种子，然后使用 rand 函数产生特定的随机发生数，使用 printf 函数将产生的随机数进行输出。

说明

如果在使用 rand 函数之前不提供种子值，也就是不用 srand 函数设定种子值，则 rand 函数总是默认以 1 作为种子，每次将产生同样的随机数序列。因此在本实例中，每次循环使用 counter+1 作为种子值。

运行程序，显示效果如图 7.7 所示。

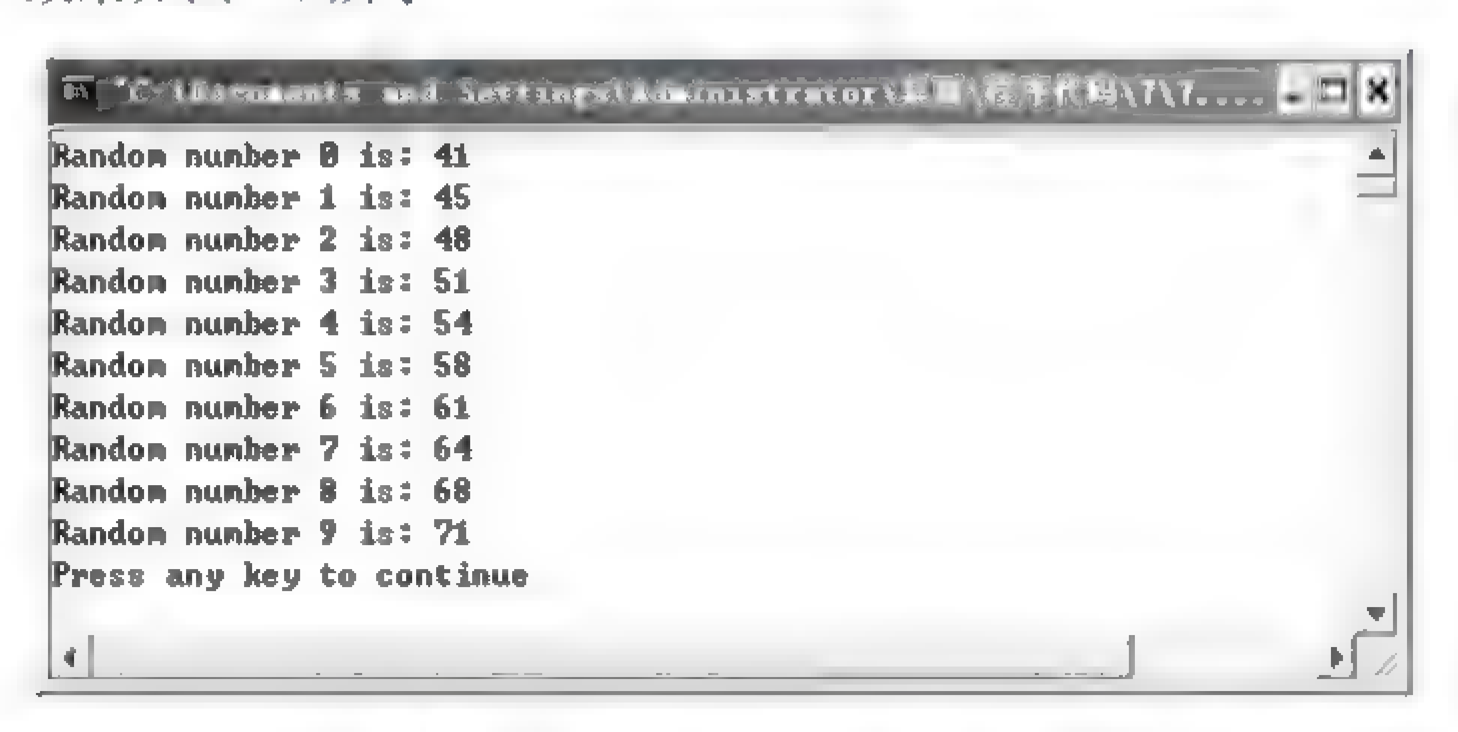


图 7.7　使用 for 语句显示随机数

for 语句的一般形式也可以使用 while 循环的形式进行表示：

```
表达式 1;
while(表达式 2)
{
    语句
    表达式 3;
}
```

上面就是使用 while 语句表示 for 语句的一般形式，其中的表达式对应着 for 语句括号中的表达式。下面通过一个实例来比较一下这两种操作。

【例 7.5】　使用 while 语句模仿 for 语句的一般形式。（**实例位置：资源包\TM\sl\7\5**）

在本实例中，先使用 for 语句实现一个有循环功能的操作，再使用 while 语句实现相同的功能。要注意实例中 for 语句中的表达式与 while 语句中的表达式所对应的位置。

```
#include<stdio.h>
```

```
int main()
{
    int iNumber;                          /*定义变量，表示 1～100 的数字*/
    int iSum=0;                           /*保存计算后的结果*/
    /*使用 for 循环*/
    for(iNumber=1;iNumber<=100;iNumber++)
    {
        iSum=iNumber+iSum;                /*累加计算*/
    }
    printf("the result is:%d\n",iSum);    /*输出计算结果*/

    iSum=0;                               /*恢复计算结果*/
    iNumber=1;                            /*设定循环控制变量的初值*/
    while(iNumber<=100)
    {
        iSum=iSum+iNumber;                /*累加计算*/
        iNumber++;                        /*循环变量自增*/
    }
    printf("the result is:%d\n",iSum);    /*输出计算结果*/
    return 0;
}
```

（1）定义变量 iNumber 表示 1～100 的数字，不过刚开始未对其进行赋值，定义变量 iSum 表示计算的结果。

（2）使用 for 语句执行循环操作，括号中第一个表达式为循环变量进行赋值。第二个表达式判断条件，条件为真，执行语句块中的内容；条件为假，不进行循环操作。

（3）在循环语句块中，进行累加运算。然后执行 for 括号中的第 3 个表达式，对循环变量进行自增操作。循环操作后，将保存有计算结果的变量 iSum 进行输出。

（4）在使用 while 语句之前要恢复变量的值。iNumber=1 就相当于 for 语句中第一个表达式的作用，为变量设置初值。然后在 while 括号中的表达式 iNumber<=100 与 for 语句中第二个表达式相对应。最后 iNumber++自加操作与 for 语句括号中的最后一个表达式相对应。

运行程序，显示效果如图 7.8 所示。

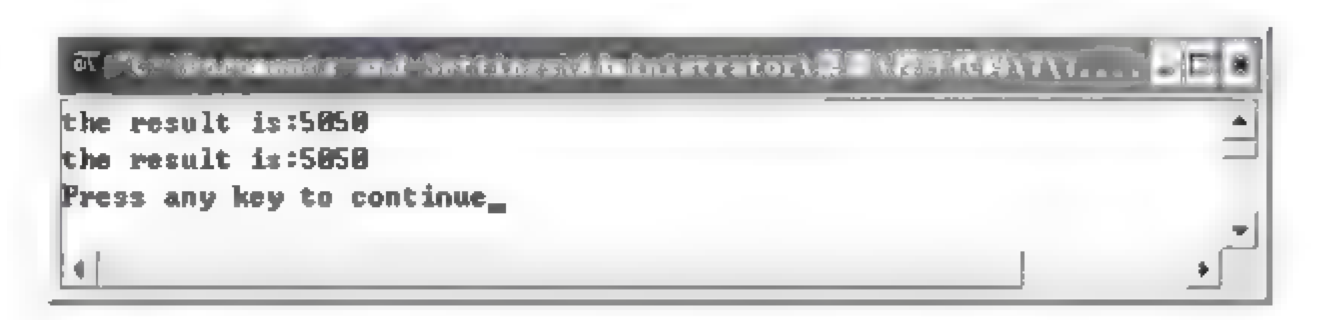


图 7.8　使用 while 语句模仿 for 语句的一般形式

7.4.2　for 循环的变体

通过上面的学习可知，for 语句的一般形式中有 3 个表达式。在实际程序的编写过程中，这 3 个表达式可以根据情况进行省略，接下来对不同情况进行讲解。

1. for 语句中省略表达式 1

for 语句中第一个表达式的作用是对循环变量设置初值。因此，如果省略了表达式 1，就会跳过这

一步操作，则应在 for 语句之前给循环变量赋值。例如：

```
for(;iNumber<10;iNumber++)
```



注意

省略表达式 1 时，其后的分号不能省略。

【例 7.6】　省略 for 语句中的第一个表达式。（**实例位置：资源包\TM\sl\7\6**）

在本实例中，同样实现 1～100 的累加计算，不过将 for 语句中第一个表达式省略。

```
#include<stdio.h>

int main()
{
    int iNumber=1;                          /*定义变量，为变量赋初始值*/
    int iSum=0;                             /*保存计算后的结果*/
    /*使用 for 循环*/
    for(;iNumber<=100;iNumber++)
    {
        iSum=iNumber+iSum;                  /*累加计算*/
    }
    printf("the result is:%d\n",iSum);      /*输出计算结果*/
    return 0;
}
```

在代码中可以看到，在 for 语句中将第一个表达式省略了，在定义 iNumber 变量时直接为其赋初值。这样在使用 for 语句循环时就不用为 iNumber 赋初值，从而省略了第一个表达式。

运行程序，显示效果如图 7.9 所示。

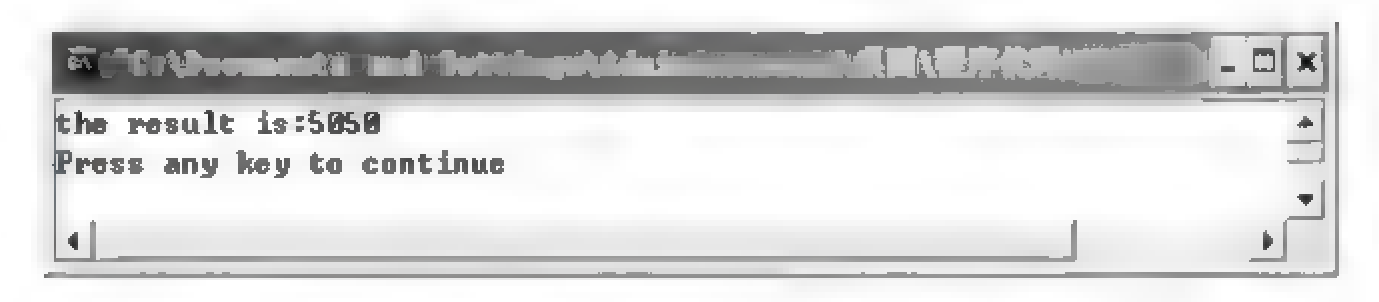


图 7.9　省略 for 语句中的第一个表达式

2．for 语句中省略表达式 2

如果表达式 2 省略，则无法判断循环条件，也即默认表达式 2 始终为真，因此循环将无终止地进行下去。例如：

```
for(iCount=1; ;iCount++)
{
    sum=sum+iCount;
}
```

在括号中，表达式 1 为赋值表达式，而表达式 2 是空缺的，这样就相当于使用了如下 while 语句：

```
iCount=1;
while(1)
```

```
{
    sum=sum+iCount;
    iCount++;
}
```

注意

如果表达式 2 为空缺，则程序将无限循环下去。

3．for 语句中省略表达式 3

表达式 3 也可以省略，但此时程序设计人员应该另外设法保证循环能正常结束，否则程序会无终止地循环下去。例如：

```
for(iCount=1;iCount<50;)
{
    sum=sum+iCount;
    iCount++;
}
```

4．3 个表达式都省略

这种情况既不设置初值，也不判断条件，也没有改变循环变量的操作，因此会无终止地执行循环体。例如：

```
for(; ;)
{
    语句
}
```

这种情况相当于 while 语句永远为真：

```
while(1)
{
    语句
}
```

5．表达式 1 为与循环变量赋值无关的表达式

表达式 1 可以是设置循环变量初值的赋值表达式，也可以是与循环无关的其他表达式。例如：

```
for(sum=0; iCount<50;iCount++)
{
    sum=sum+iCount;
}
```

7.4.3 for 语句中的逗号应用

在 for 语句中，表达式 1 和表达式 3 处除了可以使用简单的表达式外，还可以使用逗号表达式，即包含一个以上的简单表达式，中间用逗号间隔。例如，在表达式 1 处为变量 iCount 和 iSum 设置初始值：

```
for(iSum=0,iCount=1;iCount<100;iCount++)
{
    iSum=iSum+iCount;
}
```

或者执行循环变量自加操作两次：

```
for(iCount=1;iCount<100;iCount++,iCount++)
{
    iSum=iSum+iCount;
}
```

表达式 1 和表达式 3 都是逗号表达式，在逗号表达式内按照自左至右顺序求解，整个逗号表达式的值为其最右边的表达式的值。例如：

```
for(iCount=1;iCount<100;iCount++,iCount++)
```

就相当于：

```
for(iCount=1;iCount<100;iCount=iCount+2)
```

【例 7.7】 计算 1～100 所有偶数的累加结果。（**实例位置：资源包\TM\sl\7\7**）

在本实例中，为变量赋初值的操作都放在 for 语句中，并且对循环变量进行两次自加操作，这样所求出的结果就是所有偶数的和。

```
#include<stdio.h>

int main()
{
    int iCount,iSum;                                    /*定义变量*/
    /*在 for 循环中，为变量赋值，对循环变量进行两次自增运算*/
    for(iSum=0,iCount=0;iCount<=100;iCount++,iCount++)
    {
        iSum=iSum+iCount;                               /*进行累加计算*/
    }
    printf("the result is:%d\n",iSum);                  /*输出结果*/
    return 0;
}
```

在程序代码中，在 for 语句中对变量 iSum、iCount 进行初始化赋值。每次循环语句执行完后进行两次 iCount++操作，最后将结果输出。

运行程序，显示效果如图 7.10 所示。



图 7.10 计算 1～100 所有偶数的累加结果



7.5　3 种循环语句的比较

前面介绍了 3 种可以执行循环操作的语句。一般情况下，这 3 种循环之间可以相互代替。下面是对这 3 种循环语句在不同情况下的比较。

☑ while 和 do…while 循环只在 while 后面指定循环条件，在循环体中应包含使循环趋于结束的语句（如 i++或者 i=i+1 等）。for 循环可以在表达式 3 中包含使循环趋于结束的操作，可以设置将循环体中的操作全部放在表达式 3 中。因此，for 语句的功能更强，凡用 while 循环能完成的功能，用 for 循环都能实现。

☑ 用 while 和 do…while 循环时，循环变量初始化的操作应在 while 和 do…while 语句之前完成；而 for 语句可以在表达式 1 中实现循环变量的初始化。

☑ while 循环、do…while 循环和 for 循环都可以用 break 语句跳出循环，用 continue 语句结束本次循环（break 和 continue 语句将在 7.7 节中进行介绍）。



7.6　循 环 嵌 套

一个循环体内又包含另一个完整的循环结构，称之为循环的嵌套。内嵌的循环中还可以再嵌套循环，这就是多层循环。不管在什么语言中，关于循环嵌套的概念都是一样的。

7.6.1　循环嵌套的结构

while 循环、do…while 循环和 for 循环之间可以互相嵌套。例如，下面几种嵌套方式都是正确的。

1．while 结构中嵌套 while 结构

```
while(表达式)
{
    语句
    while(表达式)
    {
        语句
    }
}
```

2．do…while 结构中嵌套 do…while 结构

```
do
{
    语句
    do
    {
```

```
        语句
    }
    while(表达式);
}
while(表达式);
```

3．for 结构中嵌套 for 结构

```
for(表达式;表达式;表达式)
{
    语句
    for(表达式;表达式;表达式)
    {
        语句
    }
}
```

4．do...while 结构中嵌套 while 结构

```
do
{
    语句
    while(表达式);
    {
        语句
    }
}
while(表达式);
```

5．do...while 结构中嵌套 for 结构

```
do
{
    语句
    for(表达式;表达式;表达式)
    {
        语句
    }
}
while(表达式);
```

以上是一些嵌套的结构方式，当然还有不同结构的循环嵌套，在此不对每一项都进行列举。读者只要将每种循环结构的方式把握好，就可以正确写出循环嵌套。

7.6.2 循环嵌套实例

本节通过实例讲解，使读者了解循环嵌套的使用方法。

【例 7.8】 使用嵌套语句输出金字塔形状。(**实例位置：资源包\TM\sl\7\8**)

在本实例中，利用嵌套循环输出金字塔形状。显示一个三角形要考虑 3 点，首先要控制输出三角形的行数，其次控制三角形的空白位置，最后是将三角形进行显示。

```
#include<stdio.h>
int main()
{
    int i, j, k;                            /*定义变量 i、j、k 为基本整型*/
    for(i = 1; i <= 5; i++)                 /*控制行数*/
    {
        for(j = 1; j <= 5-i; j++)           /*空格数*/
            printf(" ");
        for(k = 1; k <= 2 *i - 1; k++)      /*显示*号的数量*/
            printf("*");
        printf("\n");
    }
    return 0;
}
```

在代码中可以看到，首先通过一个循环控制三角形的行数，也就是三角形的高度；然后在循环中嵌套循环语句，使得每一行都控制输出的空白和“*”号的数量，这样就可以将整个金字塔的形状进行输出，效果如图 7.11 所示。



图 7.11　使用嵌套语句输出金字塔形状

技巧

显示三角形时，可以想象成先显示一个倒立的直角三角形（由空格组成），然后再输出一个正立的三角形。

【例 7.9】　打印乘法口诀表。（**实例位置：资源包\TM\sl\7\9**）

本实例要求打印出乘法口诀表，在乘法口诀表中有行和列项的相乘得出的乘法结果。根据这个特点，使用循环嵌套将其显示。

```
#include<stdio.h>
int main()
{
    int iRow, iColumn;                                  /*iRow 为行，iColum 为列*/
    for(iRow = 1; iRow <= 9; iRow++)                    /*for 循环 iRow 为乘法口诀表中的行数*/
    {
        for(iColumn = 1; iColumn <= iRow; iColumn++)    /*根据 iRow，iColum 取值循环计算*/
```

```
            {
                printf("%d*%d=%d ", iRow,iColumn,iRow *iColumn);     /*输出结果*/
            }
            printf("\n");                                              /*进行换行*/
        }
        return 0;
}
```

在本实例中用到两次 for 循环，第一个 for 循环可看成乘法口诀表的行数，同时也是每行进行乘法运算的第一个因子；第二个 for 循环范围的确定建立在第一个 for 循环的基础上，即第二个 for 循环的最大取值是第一个 for 循环中变量的值。

运行程序，显示效果如图 7.12 所示。

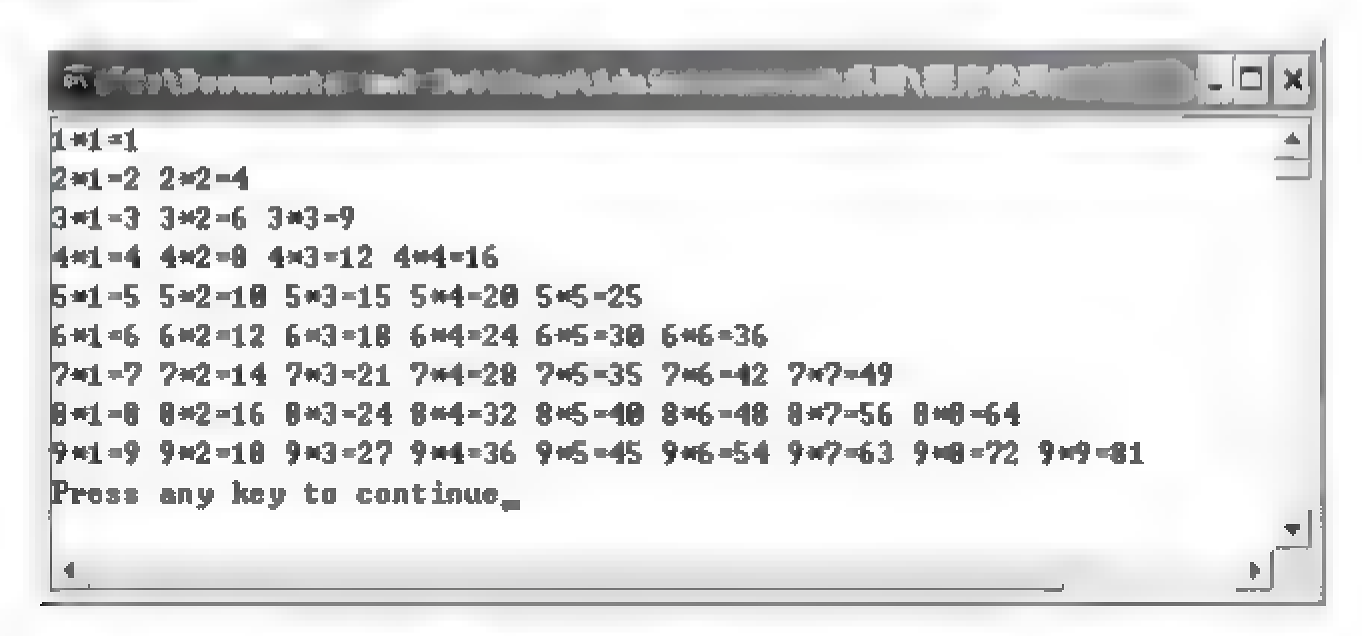


图 7.12 打印乘法口诀表

7.7 转移语句



转移语句包括 goto 语句、break 语句和 continue 语句。通过这 3 种语句，程序的执行流程会发生一定的转移。下面将对这 3 种语句的使用方式进行详细介绍。

7.7.1 goto 语句

goto 语句为无条件转移语句，可以使程序立即跳转到函数内部的任意一条可执行语句。goto 关键字后面带一个标识符，该标识符是同一个函数内某条语句的标号。标号可以出现在任何可执行语句的前面，并且以一个冒号“:”作为后缀。goto 语句的一般形式如下：

```
goto 标识符;
```

goto 后的标识符就是要跳转的目标，当然这个标识符要在程序的其他位置给出，并且其标识符要位于函数内部。函数的内容将会在后面章节进行介绍，在此对其简单了解即可。例如：

```
goto Show;
printf("the message before ShowMessage");
Show:
    printf("ShowMessage");
```

在上述代码中，goto 后的 Show 为跳转的标识符，而其后的“Show:”代码表示 goto 语句要跳转的位置。这样在上面的语句中第一个 printf 函数将不会被执行，而会执行第二个 printf 函数。

注意

跳转的方向可以向前，也可以向后；可以跳出一个循环，也可以跳入一个循环。

【例 7.10】 使用 goto 语句从循环内部跳出。（**实例位置：资源包\TM\sl\7\10**）

本实例要求在执行循环操作的过程中，当用户输入退出指令后，程序跳转到循环外部，执行程序退出前的显示操作。

```
#include<stdio.h>

int main()
{
	int iStep;							/*定义变量，表示外部循环步骤*/
	int iSelect;						/*保存用户的输入选项*/
	for(iStep=1;iStep<10;iStep++)		/*外部步骤循环*/
	{
		printf("The Step is:%d\n",iStep);	/*将其循环的步骤号显示*/
		do								/*使用 do...while 语句进行循环*/
		{
			printf("enter a number to select\n");	/*输出提示信息*/
			printf("(0 is quit,99 for the next step)\n");
			scanf("%d",&iSelect);				/*用户输入选择*/
			if(iSelect==0)						/*判断输入的是否为 0*/
			{
				goto exit;						/*执行 goto 跳转语句*/
			}
		}
		while(iSelect!=99);				/*判断用户输入*/
	}
exit:									/*跳转语句执行位置*/
	printf("Exit the program!");		/*显示程序结束信息*/
	return 0;
}
```

（1）程序运行时，for 循环控制程序步骤，程序输出的循环步骤为 1。信息提示输入数字，其中 0 表示退出，99 表示下一个步骤。

（2）在 for 循环中使用 do…while 语句判断用户输入，当条件为假时，循环结束并执行 for 循环的下一步。在程序中假如用户输入数字 3，既不退出也不到下一步骤，程序显示继续输入数字。当输入数字为 99 时，跳转到下一步，显示提示信息“The Step is:2”。

（3）如果用户输入的是 0，那么通过 if 语句判断为真，执行其中的 goto 语句进行跳转，其中 exit 为跳转的标识符。循环的外部使用 exit:表示 goto 跳转的位置。通过输出一段信息表示程序结果。

运行程序，显示效果如图 7.13 所示。

```
The Step is:1
enter a number to select
(0 is quit,99 for the next step)
3
enter a number to select
(0 is quit,99 for the next step)
99
The Step is:2
enter a number to select
(0 is quit,99 for the next step)
0
Exit the program!
Press any key to continue
```

图 7.13　使用 goto 语句从循环内部跳出

7.7.2　break 语句

有时会遇到这样的情况，不管表达式检验的结果如何，都需要强行终止循环，这时可以使用 break 语句。break 语句终止并跳出循环，继续执行后面的代码。break 语句的一般形式如下：

```
break;
```

break 语句不能用于除循环语句和 switch 语句之外的任何其他语句中。例如，在 while 循环语句中使用 break 语句：

```
while(1)
{
    printf("Break");
    break;
}
```

在上述代码中，while 语句是一个条件永远为真的循环，但由于在其中使用了 break 语句，使得程序流程跳出循环。

注意

这个 break 语句和 switch...case 分支结构中的 break 语句的作用是不同的。

【例 7.11】　使用 break 语句跳出循环。（**实例位置：资源包\TM\sl\7\11**）

使用 for 语句执行循环输出 10 次的操作，在循环体中判断输出的次数。当循环变量为 5 次时，使用 break 语句跳出循环，终止循环输出操作。

```
#include<stdio.h>

int main()
{
    int iCount;                                    /*循环控制变量*/
    for(iCount=0;iCount<10;iCount++)              /*执行 10 次循环*/
    {
        if(iCount==5)                              /*判断条件，如果 iCount 等于 5 跳出*/
```

```
        {
            printf("Break here\n");
            break;                              /*跳出循环*/
        }
        printf("the counter is:%d\n",iCount);   /*输出循环的次数*/
    }
    return 0;
}
```

变量 iCount 在 for 语句中被赋值为 0，因为 iCount<10，所以循环执行 10 次。在循环语句中使用 if 语句判断当前 iCount 的值。当 iCount 值为 5 时，if 判断为真，使用 break 语句跳出循环。

运行程序，显示效果如图 7.14 所示。

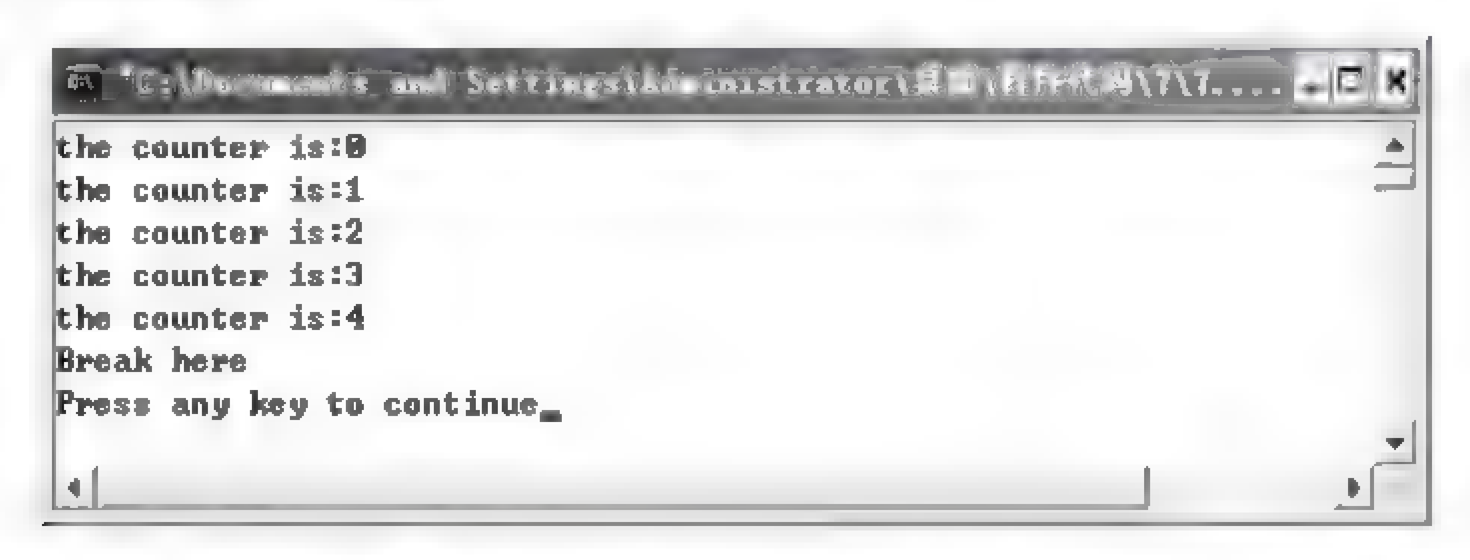


图 7.14　使用 break 语句跳出循环

7.7.3　continue 语句

在某些情况下，程序需要返回到循环头部继续执行，而不是跳出循环。continue 语句的一般形式如下：

```
continue;
```

其作用就是结束本次循环，即跳过循环体中尚未执行的部分，直接执行下一次的循环操作。

注意

continue 语句和 break 语句的区别是：continue 语句只结束本次循环，而不是终止整个循环的执行；break 语句则是结束整个循环过程，不再判断执行循环的条件是否成立。

【例 7.12】　使用 continue 语句结束本次的循环操作。（**实例位置：资源包\TM\sl\7\12**）

本实例与使用 break 语句结束循环的实例相似，区别在于将使用 break 语句的位置改写成了 continue。因为 continue 语句只结束本次循环，所以剩下的循环还是会继续执行。

```
#include<stdio.h>

int main()
{
    int iCount;                         /*循环控制变量*/
    for(iCount=0;iCount<10;iCount++)    /*执行 10 次循环*/
    {
        if(iCount==5)                   /*判断条件，如果 iCount 等于 5 则跳出*/
```

```
            {
                printf("Continue here\n");
                continue;                                   /*跳出本次循环*/
            }
            printf("the counter is:%d\n",iCount);           /*输出循环的次数*/
        }
        return 0;
}
```

通过程序的显示结果，可以看到在 iCount 等于 5 时，调用 continue 语句使得本次的循环结束。但是循环本身并没有结束，因此程序会继续执行。

运行程序，显示效果如图 7.15 所示。



图 7.15　使用 continue 语句结束本次的循环操作

7.8　小　　结

本章介绍了有关循环语句的内容，其中包括 while 结构、do...while 结构和 for 结构。

了解这些结构的使用方法，可以在程序功能上节约很多时间，无须再一条一条地进行操作。通过对 3 种循环语句的比较，可以了解它们在使用上的区别，也可以发现三者的共同之处。最后介绍了有关转移语句的内容。转移语句可以使程序设计更为灵活，使用 continue 语句可以结束本次循环操作而不终止整个循环，使用 break 语句可以结束整体循环过程，使用 goto 语句可以跳转到函数体内的任何位置。

7.9　实践与练习

1. 要求使用 for 循环打印出大写字母的 ASCII 码对照表。(**答案位置：资源包\TM\sl\7\13**)

2. 输出 0～100 不能被 3 整除的数。提示：使用 for 语句进行循环检查操作，使用 continue 语句结束不符合条件的情况。(**答案位置：资源包\TM\sl\7\14**)

第2篇

核心技术

本篇介绍了C语言中数组、函数和指针这三大部分内容，并将前面所学的基础内容融入其中，学习更高级的程序设计知识。读者通过学习本篇知识，能够编写一些简单的C语言应用程序。

第 8 章

数组

（视频讲解：1 小时 7 分钟）

在编写程序的过程中，经常会遇到需要使用很多数据量的情况。处理每个数据量都要有一个相对应的变量，如果每个变量都要单独进行定义，编程过程则会变得极其烦琐。使用数组就可以很好地解决这个问题。

本章致力于使读者掌握一维数组和二维数组的作用，并且能利用所学知识解决一些实际问题；掌握字符数组的使用及其相关操作；通过一维数组和二维数组了解有关多维数组的内容；最后利用数组应用于排序算法，并介绍有关字符串处理函数的使用。

通过阅读本章，您可以：

- 掌握一维数组和二维数组的定义和引用
- 熟悉字符数组的方式
- 了解多维数组的概念
- 掌握数组的排序算法
- 熟悉字符串处理函数的使用



8.1　一 维 数 组

8.1.1　一维数组的定义和引用

1．一维数组的定义

一维数组用以存储一维数列中数据的集合。其一般形式如下：

```
类型说明符 数组标识符[常量表达式];
```

☑ 类型说明符表示数组中所有元素的类型。
☑ 数组标识符表示该数组型变量的名称，命名规则与变量名一致。
☑ 常量表达式定义了数组中存放的数据元素的个数，即数组长度。例如 iArray[5]，5 表示数组中有 5 个元素，下标从 0 开始，到 4 结束。

例如，定义一个数组：

```
int iArray[5];
```

代码中的 int 为数组元素的类型，而 iArray 表示的是数组变量名，括号中的 5 表示的是数组中包含的元素个数。

注意

在数组 iArray[5]中只能使用 iArray[0]、iArray[1]、iArray[2]、iArray[3]、iArray[4]，而不能使用 iArray[5]。若使用 iArray[5]，则会出现下标越界的错误。

2．一维数组的引用

数组定义完成后，就要使用该数组。可以通过引用数组元素的方式使用该数组中的元素。

数组元素的一般表示形式如下：

```
数组标识符[下标]
```

例如，引用一个数组变量 iArray 中的第 3 个变量：

```
iArray[2];
```

iArray 是数组变量的名称，2 为数组的下标。有的读者会问：为什么引用第 3 个数组元素使用的数组下标是 2 呢？前面介绍过，数组的下标是从 0 开始的，也就是说下标为 2 表示的是第 3 个数组元素。

说明

下标可以是整型常量或整型表达式。

【例 8.1】　使用数组保存数据。（**实例位置：资源包\TM\sl\8\1**）

在本实例中，使用数组保存用户输入的数据，当输入完毕后逆向输出数据。

```
#include<stdio.h>

int main()
{
    int iArray[5], index, temp;                    /*定义数组及变量为基本整型*/
    printf("Please enter a Array:\n");

    for(index= 0; index< 5; index++)               /*逐个输入数组元素*/
     {
         scanf("%d", &iArray[index]);
     }

    printf("Original Array is:\n");
    for(index = 0; index< 5; index++)              /*显示数组中的元素*/
     {
         printf("%d ", iArray[index]);
     }
    printf("\n");

    for(index= 0; index < 2; index++)              /*将数组中元素的前后位置互换*/
    {
        temp = iArray[index];                      /*元素位置互换的过程借助中间变量 temp*/
        iArray[index] = iArray[4-index];
        iArray[4-index] = temp;
    }
    printf("Now Array is:\n");
    for(index = 0; index< 5; index++)              /*将转换后的数组再次输出*/
     {
         printf("%d ", iArray[index]);
     }
    printf("\n");
     return 0;
}
```

在本实例中，变量 temp 用来实现数据间的转换，而 index 用于控制循环的变量。通过语句 int iArray[5] 定义一个有 5 个元素的数组，程序中用到的 iArray[i]就是对数组元素的引用。

运行程序，显示效果如图 8.1 所示。

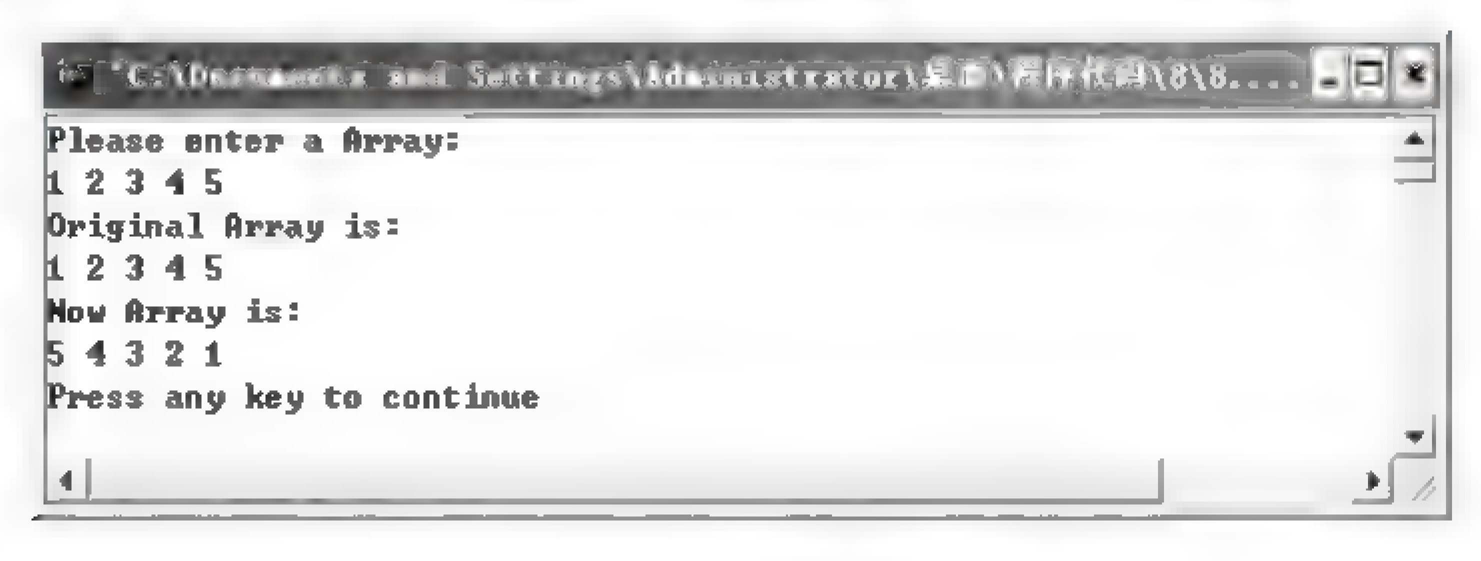


图 8.1　使用数组保存数据

8.1.2　一维数组初始化

对一维数组的初始化，可以用以下 3 种方法实现。

（1）在定义数组时直接对数组元素赋初值。例如：

```
int i,iArray[6]={1,2,3,4,5,6};
```

该方法是将数组中的元素值一次放在一对花括号中。经过上面的定义和初始化之后，数组中的元素 iArray[0]=1，iArray[1]=2，iArray[2]=3，iArray[3]=4，iArray[4]=5，iArray[5]=6。

【例 8.2】　初始化一维数组。（**实例位置：资源包\TM\sl\8\2**）

在本实例中，对定义的数组变量进行初始化操作，然后隔位进行输出。

```
#include<stdio.h>

int main()
{
    int index;                              /*定义循环控制变量*/
    int iArray[6]={0,1,2,3,4,5};            /*对数组中的元素赋值*/

    for(index=0;index<6;index+=2)           /*隔位输出数组中的元素*/
    {
        printf("%d\n",iArray[index]);
    }
    return 0;
}
```

在程序中，定义一个数组变量 iArray，并且对其进行初始化赋值。使用 for 循环输出数组中的元素，在循环中，控制循环变量使其每次增加 2，这样根据下标进行输出时就会得到隔一个元素输出的效果了。

运行程序，显示效果如图 8.2 所示。



图 8.2　初始化一维数组

（2）只给一部分元素赋值，未赋值的部分元素值为 0。

第二种为数组初始化的方式是对其中一部分元素进行赋值，例如：

```
int iArray[6]={0,1,2};
```

数组变量 iArray 包含 6 个元素，不过在初始化时只给出了 3 个值。于是数组中前 3 个元素的值对应括号中给出的值，在数组中没有得到值的元素被默认赋值为 0。

【例 8.3】　赋值数组中的部分元素。（**实例位置：资源包\TM\sl\8\3**）

在本实例中，定义数组并且为其进行初始化赋值，但只为一部分元素赋值，然后将数组中的所有

元素进行输出，观察输出的元素数值。

```
#include<stdio.h>

int main()
{
    int index;
    int iArray[6]={1,2,3};                    /*对数组中的部分元素赋初值*/

    for(index=0;index<6;index++)              /*输出数组中的所有元素*/
    {
        printf("%d\n",iArray[index]);
    }
    return 0;
}
```

在程序代码中，可以看到为数组部分元素初始化的操作和为数组元素全部赋值的操作是相同的，只不过在括号中给出的元素数值比数组元素数量少。

运行程序，显示效果如图 8.3 所示。

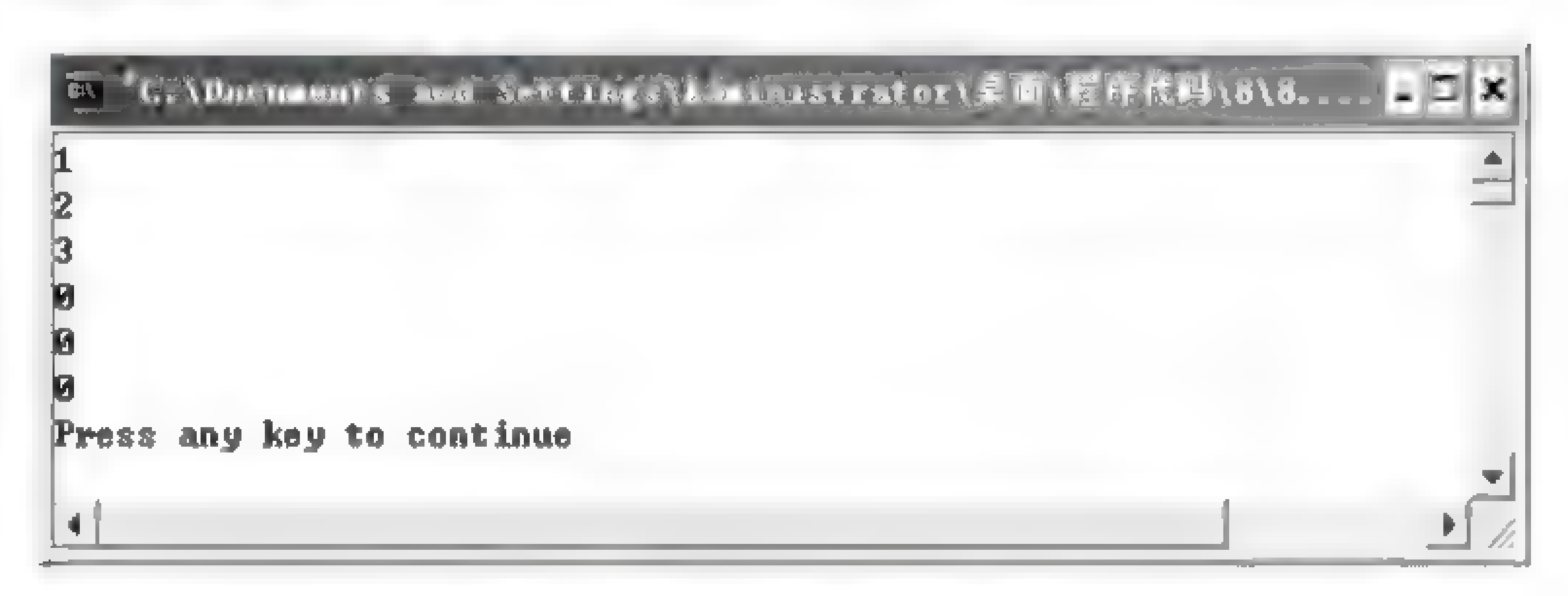


图 8.3　赋值数组中的部分元素

（3）在对全部数组元素赋初值时可以不指定数组长度。

之前在定义数组时，都在数组变量后指定了数组的元素个数。C 语言还允许在定义数组时不必指定长度，例如：

```
int iArray[]={1,2,3,4};
```

上述代码的大括号中有 4 个元素，系统就会根据给定的初始化元素值的个数来定义数组的长度，因此该数组变量的长度为 4。

注意

如果在定义数组时加入定义的长度为 10，就不能使用省略数组长度的定义方式，而必须写成：

```
int iArray[10]={1,2,3,4};
```

【例 8.4】　不指定数组的元素个数。（**实例位置：资源包\TM\sl\8\4**）

在本实例中，定义数组变量时不指定数组的元素个数，直接对其进行初始化操作，然后将其中的元素值进行输出显示。

```
#include<stdio.h>

int main()
{
    int index;
    int iArray[]={1,2,3,4,5};                    /*不指定元素个数进行初始化*/
    for(index=0;index<5;index++)
    {
        printf("%d\n",iArray[index]);             /*使用 for 循环输出数组中的所有元素*/
    }
    return 0;
}
```

运行程序，显示效果如图 8.4 所示。

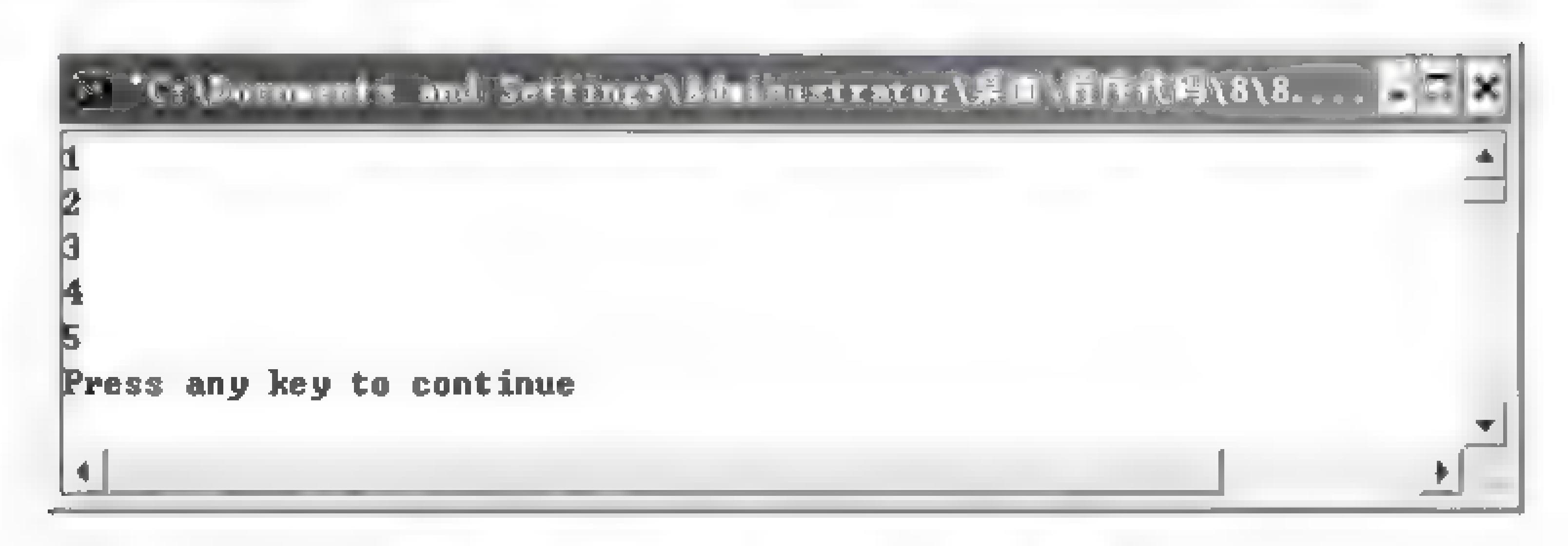


图 8.4　不指定数组的元素个数

8.1.3　一维数组的应用

例如，在一个学校的班级中会有很多学生，此时就可以使用数组来保存这些学生的姓名，以便进行管理。

【例 8.5】　使用数组保存学生姓名。（**实例位置：资源包\TM\sl\8\5**）

在本实例中，要使用数组保存学生的姓名，那么数组中的每一个元素都应该是可以保存字符串的类型，这里使用字符指针类型。

```
#include<stdio.h>

int main()
{
    char* ArrayName[5];                          /*字符指针数组*/
    int index;                                   /*循环控制变量*/
    ArrayName[0]="WangJiasheng";                 /*为数组元素赋值*/
    ArrayName[1]="LiuWen";
    ArrayName[2]="SuYuqun";
    ArrayName[3]="LeiYu";
    ArrayName[4]="ZhangMeng";
    for(index=0;index<5;index++)                 /*使用循环显示名称*/
    {
        printf("%s\n",ArrayName[index]);
```

```
    }

    return 0;
}
```

从上述程序代码可以看出，char* ArrayName[5]定义了一个具有 5 个字符指针元素的数组，然后利用每个元素保存一个学生的姓名，使用 for 循环将其数组中保存的姓名数据进行输出。

运行程序，显示效果如图 8.5 所示。

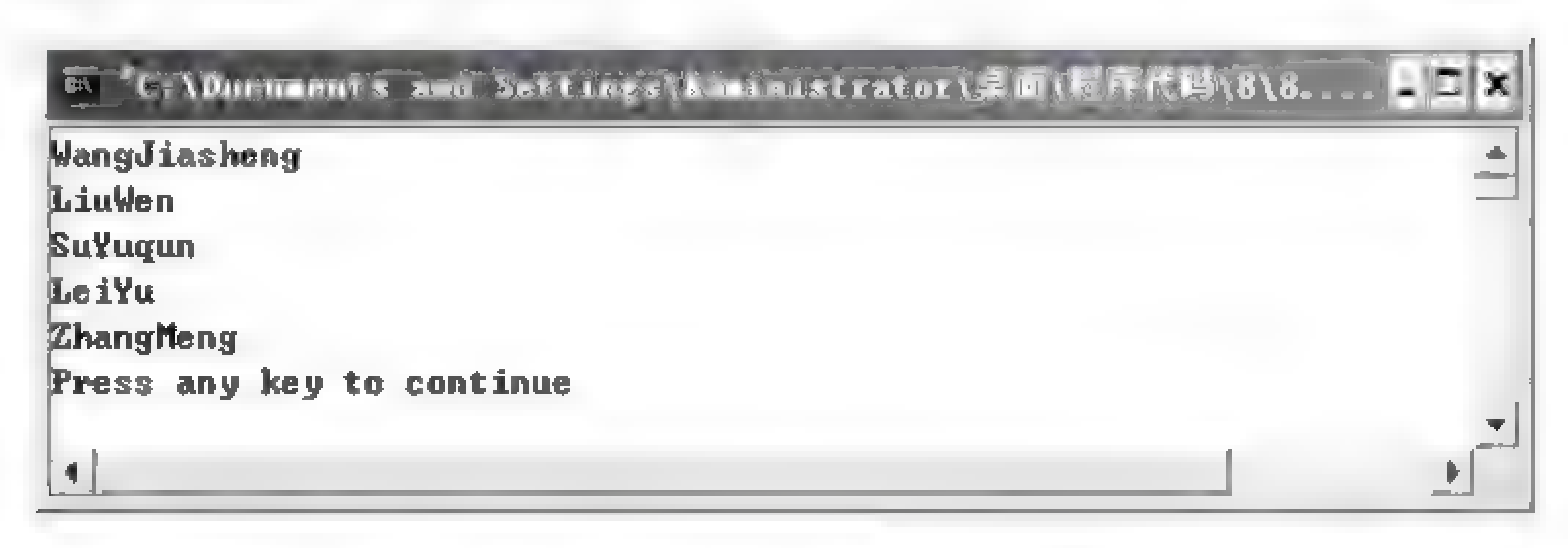


图 8.5　使用数组保存学生姓名



8.2　二维数组

8.2.1　二维数组的定义和引用

1．二维数组的定义

二维数组的声明与一维数组相同，一般形式如下：

```
数据类型  数组名[常量表达式 1][常量表达式 2];
```

其中，“常量表达式 1”被称为行下标，“常量表达式 2”被称为列下标。如果有二维数组 array[n][m]，则二维数组的下标取值范围如下：

☑　行下标的取值范围为 0～n-1。

☑　列下标的取值范围为 0～m-1。

☑　二维数组的最大下标元素是 array[n-1][m-1]。

例如，定义一个 3 行 4 列的整型数组：

```
int array[3][4];
```

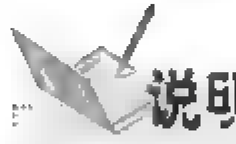
上述代码声明了一个 3 行 4 列的数组，数组名为 array，其下标变量的类型为整型。该数组的下标变量共有 3×4 个，即 array[0][0]、array[0][1]、array[0][2]、array[0][3]、array[1][0]、array[1][1]、array[1][2]、array[1][3]、array[2][0]、array[2][1]、array[2][2]、array[2][3]。

在 C 语言中，二维数组是按行排列的，即按行顺次存放，先存放 array[0]行，再存放 array[1]行。每行有 4 个元素，也是依次存放。

2. 二维数组的引用

二维数组元素的引用一般形式如下：

```
数组名[下标][下标];
```



说明

二维数组的下标可以是整型常量或整型表达式。

例如，对一个二维数组的元素进行引用：

```
array[1][2];
```

上述代码表示的是对 array 数组中第 2 行的第 3 个元素进行引用。



注意

不管是行下标还是列下标，其索引都是从 0 开始的。

这里和一维数组一样，要注意下标越界的问题。例如：

```
int array[2][4];
...                                    /*对数组元素进行赋值*/
array[2][4]=9;                         /*错误！*/
```

上述代码的表示是错误的。

首先 array 为 2 行 4 列的数组，那么它的行下标的最大值为 1，列下标的最大值为 3，所以 array[2][4] 超过了数组的范围，下标越界。

8.2.2　二维数组初始化

二维数组和一维数组一样，也可以在声明时对其进行初始化。在给二维数组赋初值时，有以下 4 种情况：

（1）可以将所有数据写在一个大括号内，按照数组元素排列顺序对元素赋值。例如：

```
int array[2][2] = {1,2,3,4};
```

如果大括号内的数据少于数组元素的个数，则系统将默认后面未被赋值的元素值为 0。

（2）在为所有元素赋初值时，可以省略行下标，但是不能省略列下标。例如：

```
int array[][3] = {1,2,3,4,5,6};
```

系统会根据数据的个数进行分配，一共有 6 个数据，而数组每行分为 3 列，当然可以确定数组为 2 行。

（3）也可以分行给数组元素赋值。例如：

```
int a[2][3] = {{1,2,3},{4,5,6}};
```

在分行赋值时，可以只对部分元素赋值。例如：

```
int a[2][3] = {{1,2},{4,5}};
```

在上行代码中，各个元素的值为：a[0][0]的值是 1；a[0][1]的值是 2；a[0][2]的值是 0；a[1][0]的值是 4；a[1][1]的值是 5；a[1][2]的值是 0。



说明

还记得前面介绍的一维数组初始化时的情况吗？如果只给一部分元素赋值，则未赋值的部分元素值为 0。

（4）二维数组也可以直接对数组元素赋值。例如：

```
int a[2][3];
a[0][0] = 1;
a[0][1] = 2;
```

这种赋值的方式就是使用数组引用的数组中的元素。

【例 8.6】 使用二维数组保存数据。（**实例位置：资源包\TM\sl\8\6**）

本实例通过键盘为二维数组元素赋值，显示二维数组，求出二维数组中最大元素和最小元素的值及其下标，将二维数组转换为另一个二维数组并显示。

```
#include<stdio.h>

int main()
{
	int a[2][3],b[3][2];				/*定义两个数组*/
	int max,min;					/*表示最大值和最小值*/
	int h,l,i,j;					/*用于控制循环*/

	for(i=0;i<2;i++)				/*通过键盘为数组元素赋值*/
	{
		for(j=0;j<3;j++)
		{
			printf("a[%d][%d]=",i,j);
			scanf("%d",&a[i][j]);
		}
	}
	printf("输出二维数组：\n");			/*信息提示*/
	for(i=0;i<2;i++)
	{
		for(j=0;j<3;j++)
		{
			printf("%d\t",a[i][j]);
		}
		printf("\n");				/*使元素分行显示*/
	}
	/*求数组中的最大元素及其下标*/
	max = a[0][0];
	h = 0;
```

```
l = 0;
for(i=0;i<2;i++)
{
        for(j=0;j<3;j++)
        {
                if(max < a[i][j])
                {
                        max = a[i][j];
                        h = i;
                        l = j;
                }
        }
}
printf("数组中最大元素是：\n");
printf("max:a[%d][%d]=%d\n",h,l,max);
/*求数组中的最小元素及其下标*/
min = a[0][0];
h = 0;
l = 0;
for(i=0;i<2;i++)
{
        for(j=0;j<3;j++)
        {
                if(min > a[i][j])
                {
                        min = a[i][j];
                        h = i;
                        l = j;
                }
        }
}
printf("数组中最小元素是：\n");
printf("min:a[%d][%d]=%d\n",h,l,min);
/*将数组 a 转换后存入数组 b 中*/
for(i=0;i<2;i++)
{
        for(j=0;j<3;j++)
        {
                b[j][i] = a[i][j];
        }
}
printf("输出转换后的二维数组：\n");
for(i=0;i<3;i++)
{
        for(j=0;j<2;j++)
        {
                printf("%d\t",b[i][j]);
        }
```

```
        printf("\n");                              /*使元素分行显示*/
    }
    return 0;
}
```

（1）在程序中根据每一次的提示，输入相应数组元素的数据，然后将这个 2 行 3 列的数组输出。在输出数组元素时，为了使输出的数据更容易观察，使用“\t”转换字符来控制间距。

（2）寻找数组中的最大数值，使用 max 变量表示最大数值，使用双重循环比较二维数组中的每一个元素，当一个元素的数值比 max 变量表示的数值大时，就将该值赋给 max 变量，然后使用 h 和 j 变量保存最大数值在数组中的下标位置。根据保存数据的变量，最后将最大值和该数据在数组中的下标都输出显示。

（3）得到数组中最小值的方法与得到最大值的方法相同。

（4）最后将数组转换成 3 行 2 列的数组，其中通过循环的控制，将一个数组中元素的数值赋值到转换后的数组中。

运行程序，显示效果如图 8.6 所示。



图 8.6　使用二维数组保存数据

8.2.3　二维数组的应用

【例 8.7】　输入任意一个 3 行 3 列的二维数组，求对角元素之和。（**实例位置：资源包\TM\sl\8\7**）

在本实例中，使用二维数组保存一个 3 行 3 列的数组，利用双重循环访问数组中的每一个元素。在循环中判断是否是对角线上的元素，然后进行累加计算。

```
#include<stdio.h>
int main()
{
    int a[3][3];                                   /*定义一个 3 行 3 列的数组*/
    int i,j,sum=0;                                 /*定义循环控制变量和保存数据变量 sum*/
    printf("please input:\n");
    for(i=0;i<3;i++)                               /*利用循环对数组元素进行赋值*/
```

```
    {
        for(j=0;j<3;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }

    for(i=0;i<3;i++)                          /*使用循环计算对角线的总和*/
    {
        for(j=0;j<3;j++)
        {
            if(i==j)
            {
                sum=sum+a[i][j];              /*进行数据的累加计算*/
            }
        }
    }
    printf("the result is :%d\n",sum);        /*输出最后的结果*/
    return 0;
}
```

运行程序，显示效果如图 8.7 所示。

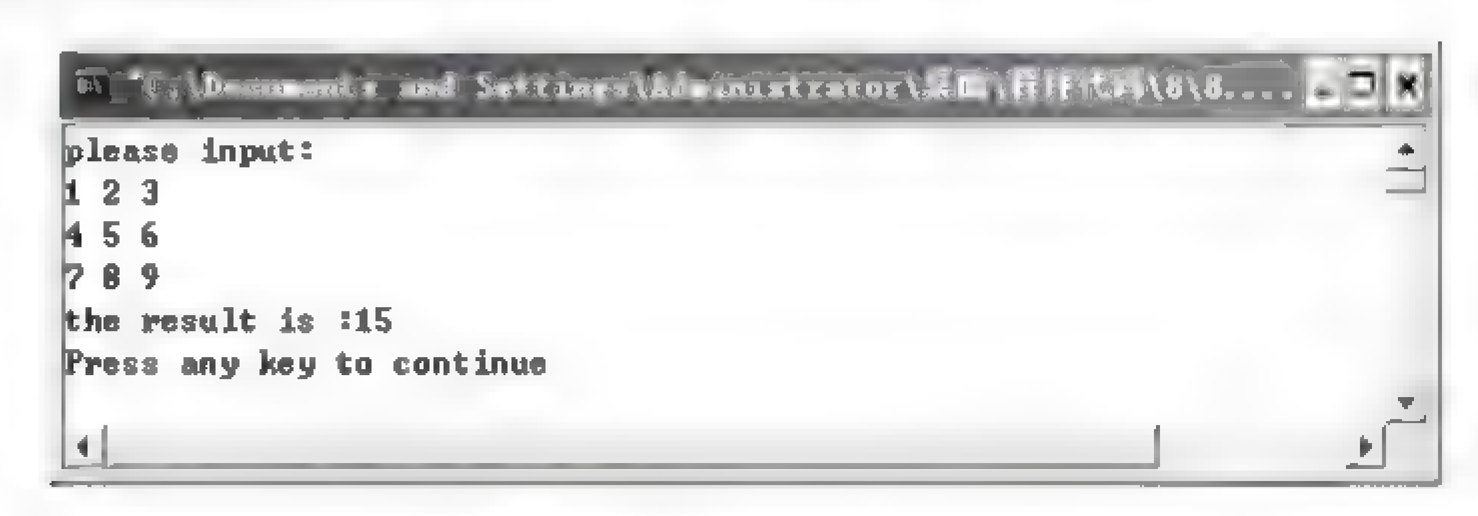


图 8.7　求对角元素之和



8.3　字符数组

数组中的元素类型为字符型时，称为字符数组。字符数组中的每个元素可以存放一个字符。字符数组的定义和使用方法与其他数据类型的数组基本相似。

8.3.1　字符数组的定义和引用

1．字符数组的定义

字符数组的定义与其他数据类型的数组定义类似，一般形式如下：

```
char 数组标识符[常量表达式]
```

因为要定义的是字符型数据，所以在数组标识符前所用的类型是 char，后面括号中表示的是数组

元素的数量。

例如，定义一个字符数组 cArray：

```
char cArray[5];
```

其中，cArray 表示数组的标识符，5 表示数组中包含 5 个字符型的变量元素。

2．字符数组的引用

字符数组的引用与其他类型数据引用一样，也是使用下标的形式。例如，引用上面定义的数组 cArray 中的元素：

```
cArray[0]='H';
cArray[1]='e';
cArray[2]='l';
cArray[3]='l';
cArray[4]='o';
```

上面的代码依次引用数组中的元素为其赋值。

8.3.2 字符数组初始化

在对字符数组进行初始化操作时，有以下几种方法。

（1）逐个字符赋给数组中的各元素。

这是最容易理解的初始化字符数组的方式。例如，初始化一个字符数组：

```
char cArray[5]={'H','e','l','l','o'};
```

定义包含 5 个元素的字符数组，在初始化的大括号中，每一个字符对应赋值一个数组元素。

【例 8.8】 使用字符数组输出一个字符串。（**实例位置：资源包\TM\sl\8\8**）

在本实例中，定义一个字符数组，通过初始化操作保存一个字符串，然后通过循环引用每一个数组元素并进行输出操作。

```
#include<stdio.h>

int main()
{
    char cArray[5]={'H','e','l','l','o'};          /*初始化字符数组*/
    int i;                                         /*循环控制变量*/
    for(i=0;i<5;i++)                               /*进行循环*/
    {
        printf("%c",cArray[i]);                    /*输出字符数组元素*/
    }
    printf("\n");                                  /*输出换行*/
    return 0;
}
```

在初始化字符数组时要注意，每一个元素的字符都是使用一对单引号“''”表示的。在循环中，因

为输出的类型是字符型，所以在 printf 函数中使用的是“%c”。通过循环变量 i，cArray[i]是对数组中每一个元素的引用。

运行程序，显示效果如图 8.8 所示。



图 8.8 使用字符数组输出一个字符串

（2）在定义字符数组时进行初始化，此时可以省略数组长度。

如果初值个数与预定的数组长度相同，在定义时可以省略数组长度，系统会自动根据初值个数来确定数组长度。例如，上面初始化字符数组的代码可以写成：

```
char cArray[]={'H','e','l','l','o'};
```

可见，代码中定义的 cArray[]中没有给出数组的大小，但是根据初值的个数可以确定数组的长度为 5。

（3）利用字符串给字符数组赋初值。

通常用一个字符数组来存放一个字符串。例如，用字符串的方式对数组作初始化赋值：

```
char cArray[]={"Hello"};
```

或者将“{}”去掉，写成：

```
char cArray[]="Hello";
```

【例 8.9】 使用二维字符数组输出一个钻石形状。（**实例位置：资源包\TM\sl\8\9**）

在本实例中定义一个二维数组，并且利用数组的初始化赋值设置钻石形状。

```
#include<stdio.h>

int main()
{
    int iRow,iColumn;                                    /*用来控制循环的变量*/
    char cDiamond[][5]={{' ',' ','*'},                   /*初始化二维字符数组*/
                        {' ','*',' ','*'},
                        {'*',' ',' ',' ','*'},
                        {' ','*',' ','*'},
                        {' ',' ','*'} };
    for(iRow=0;iRow<5;iRow++)                            /*利用循环输出数组*/
    {
        for(iColumn=0;iColumn<5;iColumn++)
        {
            printf("%c",cDiamond[iRow][iColumn]);        /*输出数组元素*/
        }
        printf("\n");                                    /*进行换行*/
    }
```

```
    return 0;
}
```

为了方便读者观察字符数组的初始化，这里将其进行对齐。在初始化时，虽然没有给出一行中具体的元素个数，但是通过初始化赋值可以确定其大小为 5，最后通过双重循环将所有数组元素输出显示。

运行程序，显示效果如图 8.9 所示。

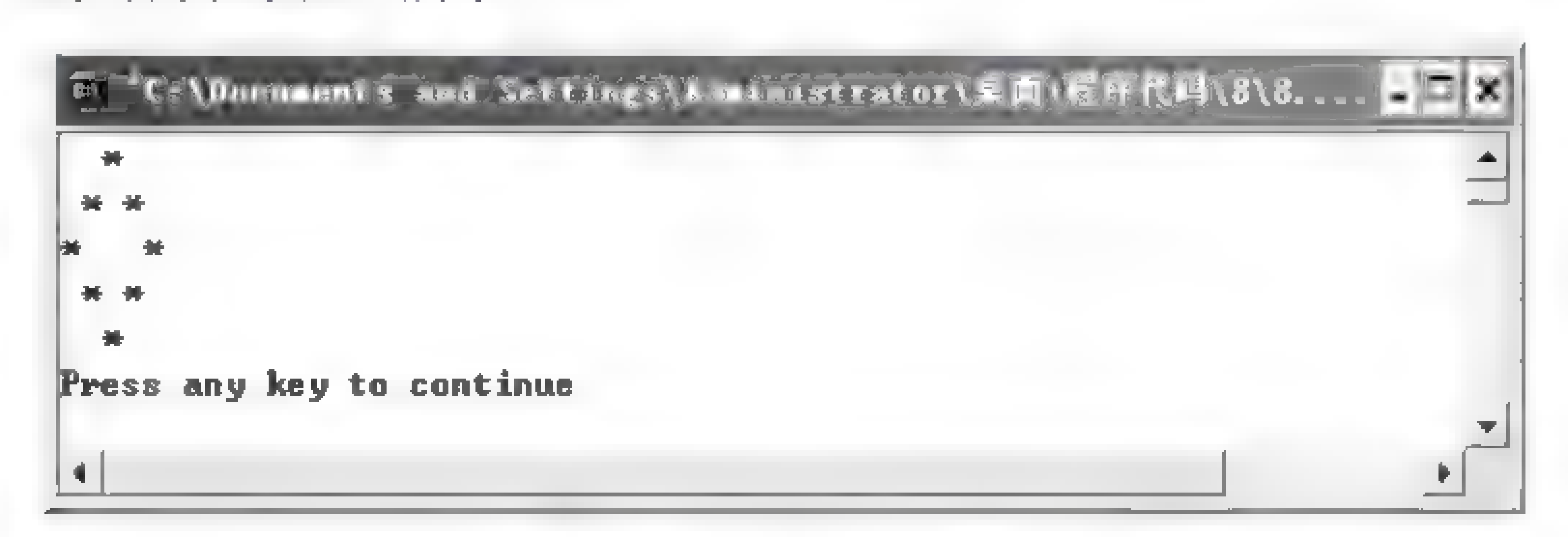


图 8.9　输出一个钻石形状

8.3.3　字符数组的结束标志

在 C 语言中，使用字符数组保存字符串，也就是使用一个一维数组保存字符串中的每一个字符，此时系统会自动为其添加“\0”作为结束符。

例如，使用下述代码可以初始化一个字符数组：

```
char cArray[]="Hello";
```

字符串总是以“\0”作为串的结束符，因此当把一个字符串存入一个数组时，也同时把结束符“\0”存入数组，并以此作为该字符串是否结束的标志。

注意

有了“\0”标志后，字符数组的长度就显得不那么重要了。当然在定义字符数组时应估计实际字符串长度，保证数组长度始终大于字符串实际长度。如果在一个字符数组中先后存放多个不同长度的字符串，则应使数组长度大于最长的字符串的长度。

用字符串方式赋值比用字符逐个赋值要多占一个字节，多占的这个字节用于存放字符串结束标志“\0”。上面的字符数组 cArray 在内存中的实际存放情况如图 8.10 所示。



H	e	l	l	o	\0

图 8.10　cArray 在内存中的实际存放情况

“\0”是由 C 编译系统自动加上的。因此上面的赋值语句等价于：

```
char cArray[]={'H','e','l','l','o','\0'};
```

字符数组并不要求最后一个字符为“\0”，甚至可以不包含“\0”。因此下面的写法也是合法的：

```
char cArray[5]={'H','e','l','l','o'};
```

是否加“\0”应根据需要来决定。由于系统对字符串常量会自动加一个“\0”，因此，为了使处理方法一致，且便于测定字符串的实际长度以及在程序中做相应的处理，在字符数组中也常常人为地加上一个“\0”。例如：

```
char cArray[6]={'H','e','l','l','o','\0'};
```

8.3.4 字符数组的输入和输出

字符数组的输入和输出有以下两种方法。

（1）使用格式符“%c”进行输入和输出。

使用格式符“%c”实现字符数组中字符的逐个输入与输出。例如，循环输出字符数组中的元素：

```
for(i=0;i<5;i++)                              /*进行循环*/
{
    printf("%c",cArray[i]);                   /*输出字符数组元素*/
}
```

其中变量为循环的控制变量，并且在循环中作为数组的下标进行循环输出。

（2）使用格式符“%s”进行输入或输出。

使用格式符“%s”将整个字符串依次输入或输出。例如，输出一个字符串：

```
char cArray[]="GoodDay!";                     /*初始化字符数组*/
printf("%s",cArray);                          /*输出字符串*/
```

其中使用格式符“%s”将字符串进行输出需要注意以下几种情况：

☑ 输出字符不包括结束符“\0”。

☑ 用“%s”格式输出字符串时，printf 函数中的输出项是字符数组名 cArray，而不是数组中的元素名 cArray[0]等。

☑ 如果数组长度大于字符串实际长度，则也只输出到“\0”为止。

☑ 如果一个字符数组中包含多个“\0”结束字符，则在遇到第一个“\0”时输出就结束。

【例 8.10】 使用两种方式输出字符串。（**实例位置：资源包\TM\sl\8\10**）

在本实例中为定义的字符数组进行初始化操作，在输出字符数组中保存的数据时，可以逐个将数组中的元素进行输出，也可以直接将字符串进行输出。

```
#include<stdio.h>
int main()
{
    int iIndex;                               /*循环控制变量*/
    char cArray[12]="MingRi KeJi";            /*定义字符数组，用于保存字符串*/

    for(iIndex=0;iIndex<12;iIndex++)
    {
        printf("%c",cArray[iIndex]);          /*逐个输出字符数组中的字符*/
    }
    printf("\n%s\n",cArray);                  /*直接将字符串输出*/
```

```
    return 0;
}
```

在代码中，对数组中的元素逐个进行输出时，使用的是循环的方式。而直接输出字符串时，利用的是 printf 函数中的格式符“%s”。要注意，直接输出字符串时，不能使用格式符“%c”。

运行程序，显示效果如图 8.11 所示。



图 8.11　使用两种方式输出字符串

8.3.5　字符数组的应用

【例 8.11】　计算字符串中单词的个数。（**实例位置：资源包\TM\sl\8\11**）

在本实例中输入一行字符，然后统计其中有多少个单词，要求每个单词之间用空格分隔开，且最后的字符不能为空格。

```
#include<stdio.h>

int main()
{
    char cString[100];                              /*定义保存字符串的数组*/
    int iIndex, iWord=1;                            /*iWord 表示单词的个数*/
    char cBlank;                                    /*表示空格*/
    gets(cString);                                  /*输入字符串*/

    if(cString[0]=='\0')                            /*判断字符串为空的情况*/
    {
        printf("There is no char!\n");
    }
    else if(cString[0]==' ')                        /*判断第一个字符为空格的情况*/
    {
        printf("First char just is a blank!\n");
    }
    else
    {
        for(iIndex=0;cString[iIndex]!='\0';iIndex++)    /*循环判断每一个字符*/
        {
            cBlank=cString[iIndex];                 /*得到数组中的字符元素*/
            if(cBlank==' ')                         /*判断是不是空格*/
            {
                iWord++;                            /*如果是则加 1*/
            }
```

```
        }
        printf("%d\n",iWord);
    }
    return 0;
}
```

按照要求使用 gets 函数将输入的字符串保存在 cString 字符数组中。首先对输入的字符进行判断，数组中的第一个输入字符如果是结束符或空格，那么进行消息提示；如果不是，则说明输入的字符串是正常的，这样就在 else 语句中进行处理。

使用 for 循环判断每一个数组中的字符是否为结束符，如果是，则循环结束；如果不是，则在循环语句中判断是否为空格，遇到一个空格，则对单词计数变量 iWord 进行自加操作。

运行程序，显示效果如图 8.12 所示。



图 8.12　计算字符串中单词的个数



8.4　多 维 数 组

多维数组的声明和二维数组相同，只是下标更多。其一般形式如下：

```
数据类型 数组名[常量表达式 1][常量表达式 2]...[常量表达式 n];
```

例如，声明多维数组：

```
int iArray1[3][4][5];
int iArray2[4][5][7][8];
```

在上面的代码中分别定义了一个三维数组 iArray1 和一个四维数组 iArray2。由于数组元素的位置都可以通过偏移量计算，因此对于三维数组 a[m][n][p]来说，元素 a[i][j][k]所在的地址是从 a[0][0][0]算起到（i*n*p+j*p+k）个单位的位置。



8.5　数组的排序算法

通过学习前面的内容，读者已经了解到了数组的理论知识。虽然数组是一组有序数据的集合，但是这里的有序指的是数组元素在数组中所处的位置，而不是根据数组元素的数值大小进行排列的。那么如何才能将数组元素按照数值的大小进行排列呢？可以通过一些排序算法来实现，本节将带领读者了解一下数组的排序算法。

8.5.1 选择法排序

选择法排序指每次选择所要排序的数组中的最大值（这里指由大到小排序，由小到大排序则应选择最小值）的数组元素，将这个数组元素的值与最前面没有进行排序的数组元素的值互换。

下面以数字 9、6、15、4、2 为例，进行选择法排序，每次交换的顺序如表 8.1 所示。

表 8.1 选择法排序

数组元素 / 排序过程	元素【0】	元素【1】	元素【2】	元素【3】	元素【4】
起始值	9	6	15	4	2
第 1 次	2	6	15	4	9
第 2 次	2	4	15	6	9
第 3 次	2	4	6	15	9
第 4 次	2	4	6	9	15
排序结果	2	4	6	9	15

可以发现，在第一次排序过程中将第一个数字和最小的数字进行了位置互换；而第二次排序过程中，将第二个数字和剩下的数字中最小的数字进行了位置互换；依此类推，每次都将下一个数字和剩余的数字中最小的数字进行位置互换，直到将一组数字按从小到大排序。

下面通过实例来看一下如何使用选择法实现数组元素从小到大的排序。

【例 8.12】 选择法排序。（**实例位置：资源包\TM\sl\8\12**）

在本实例中，声明了一个整型数组和两个整型变量，其中整型数组用于存储用户输入的数字，而整型变量用于存储数值最小的数组元素的数值和该元素的位置，然后通过双层循环进行选择法排序，最后将排序好的数组进行输出。

```
#include <stdio.h>
int main()
{
	int i,j;
	int a[10];
	int iTemp;
	int iPos;
	printf("为数组元素赋值：\n");
	/*从键盘为数组元素赋值*/
	for(i=0;i<10;i++)
	{
		printf("a[%d]=",i);
		scanf("%d", &a[i]);
	}

	/*从小到大排序*/
	for(i=0;i<9;i++)						/*设置外层循环为下标 0~8 的元素*/
	{
```

```
            iTemp = a[i];                          /*设置当前元素为最小值*/
            iPos = i;                              /*记录元素位置*/
            for(j=i+1;j<10;j++)                    /*内层循环 i+1 到 9*/
            {
                if(a[j]<iTemp)                     /*如果当前元素比最小值还小*/
                {
                    iTemp = a[j];                  /*重新设置最小值*/
                    iPos = j;                      /*记录元素位置*/
                }
            }
            /*交换两个元素值*/
            a[iPos] = a[i];
            a[i] = iTemp;
        }

        /*输出数组*/
        for(i=0;i<10;i++)
        {
            printf("%d\t",a[i]);                   /*输出制表位*/
            if(i == 4)                             /*如果是第 5 个元素*/
                printf("\n");                      /*输出换行*/
        }

        return 0;                                  /*程序结束*/
}
```

（1）声明一个整型数组，并通过键盘为数组元素赋值。

（2）设置一个嵌套循环，第一层循环为前 9 个数组元素，并在每次循环时将对应当前次数的数组元素设置为最小值（如果当前是第 3 次循环，那么将数组中第 3 个元素（也就是下标为 2 的元素）设置为当前的最小值）；在第二层循环中，循环比较该元素之后的各个数组元素，并将每次比较结果中较小的数设置为最小值，在第二层循环结束时，将最小值与开始时设置为最小值的数组元素进行互换。当所有循环都完成以后，就将数组元素按照从小到大的顺序重新排列了。

（3）循环输出数组中的元素，并在输出 5 个元素以后进行换行，在下一行输出后面的 5 个元素。

运行程序，显示效果如图 8.13 所示。



图 8.13　选择法排序

8.5.2 冒泡法排序

冒泡法排序指的是在排序时，每次比较数组中相邻的两个数组元素的值，将较小的数（从小到大排列）排在较大的数前面。

下面仍以数字 9、6、15、4、2 为例，对这几个数字进行冒泡法排序。每次排序后的顺序如表 8.2 所示。

表 8.2 冒泡法排序

数组元素 / 排序过程	元素【0】	元素【1】	元素【2】	元素【3】	元素【4】
起始值	9	6	15	4	2
第 1 次	2	9	6	15	4
第 2 次	2	4	9	6	15
第 3 次	2	4	6	9	15
第 4 次	2	4	6	9	15
排序结果	2	4	6	9	15

可以发现，在第一次排序过程中，将最小的数字移动到第一的位置，并将其他数字依次向后移动；在第二次排序过程中，从第二个数字开始的剩余数字中选择最小的数字，并将其移动到第二的位置，剩余数字依次向后移动；依此类推，每次都将剩余数字中的最小数字移动到当前剩余数字的最前方，直到将一组数字按从小到大排序为止。

下面通过实例来看一下如何使用冒泡法排序实现数组元素从小到大的排序。

【例 8.13】 冒泡法排序。（**实例位置：资源包\TM\sl\8\13**）

在本实例中，声明了一个整型数组和一个整型变量，其中整型数组用于存储用户输入的数字，而整型变量则作为两个元素交换时的中间变量，通过双层循环进行冒泡法排序，最后将排好序的数组进行输出。

```
#include<iostream.h>
int main()
{
	int i,j;
	int a[10];
	int iTemp;
	printf("为数组元素赋值：\n");
	/*通过键盘为数组元素赋值*/
	for(i=0;i<10;i++)
	{
		printf("a[%d]=",i);
		scanf("%d", &a[i]);
	}

	/*从小到大排序*/
```

```
    for(i=1;i<10;i++)                /*外层循环元素下标为 1~9*/
    {
        for(j=9;j>=i;j--)            /*内层循环元素下标为 i~9*/
        {
            if(a[j]<a[j-1])          /*如果前一个数比后一个数大*/
            {
                /*交换两个数组元素的值*/
                iTemp = a[j-1];
                a[j-1] = a[j];
                a[j] = iTemp;
            }
        }
    }

    /*输出数组*/
    for(i=0;i<10;i++)
    {
        printf("%d\t",a[i]);         /*输出制表位*/
        if(i == 4)                   /*如果是第 5 个元素*/
            printf("\n");            /*输出换行*/
    }

    return 0;                        /*程序结束*/
}
```

（1）声明一个整型数组，并通过键盘为数组元素赋值。

（2）设置一个嵌套循环，第一层循环为后 9 个数组元素。在第二层循环中，从最后一个数组元素开始向前循环，直到前面第一个没有进行排序的数组元素。循环比较这些数组元素，如果在比较中后一个数组元素的值小于前一个数组元素的值，则将两个数组元素的值进行互换。当所有循环都完成以后，就将数组元素按照从小到大的顺序重新排列了。

（3）循环输出数组中的元素，并在输出 5 个元素以后进行换行，在下一行输出后面的 5 个元素。

运行程序，显示效果如图 8.14 所示。



图 8.14　冒泡法排序

8.5.3　交换法排序

交换法排序是将每一位数与其后的所有数一一比较，如果发现符合条件的数据，则交换数据。首

先，用第一个数依次与其后的所有数进行比较，如果存在比其值大（小）的数，则交换这两个数，继续向后比较其他数直至最后一个数。然后再使用第二个数与其后面的数进行比较，如果存在比其值大（小）的数，则交换这两个数。继续向后比较其他数，直至最后一个数比较完成。

下面以数字 9、6、15、4、2 为例，进行交换法排序。每次排序后的顺序如表 8.3 所示。

表 8.3　交换法排序

数组元素 / 排序过程	元素【0】	元素【1】	元素【2】	元素【3】	元素【4】
起始值	9	6	15	4	2
第 1 次	2	9	15	6	4
第 2 次	2	4	15	9	6
第 3 次	2	4	6	15	9
第 4 次	2	4	6	9	15
排序结果	2	4	6	9	15

可以发现，在第一次排序过程中将第一个数与后边的数依次进行比较。首先比较 9 和 6，9 大于 6，交换两个数的位置，然后数字 6 成为第一个数字；用 6 和第 3 个数字 15 进行比较，6 小于 15，保持原来的位置；然后用 6 和 4 进行比较，6 大于 4，交换两个数字的位置；再用当前数字 4 与最后的数字 2 进行比较，4 大于 2，则交换两个数字的位置，从而得到表 8.3 中第一次的排序结果。然后使用相同的方法，从当前第二个数字 9 开始，继续和后面的数字进行比较，如果遇到比当前数字小的数字则交换位置，依此类推，直到将一组数字按从小到大排序为止。

下面通过实例来看一下如何通过交换法实现数组元素从小到大的排序。

【例 8.14】　交换法排序。（**实例位置：资源包\TM\sl\8\14**）

在本实例中，声明了一个整型数组和一个整型变量，其中整型数组用于存储用户输入的数字，而整型变量则作为两个元素交换时的中间变量，然后通过双层循环进行交换法排序，最后将排好序的数组进行输出。

```
#include<stdio.h>
int main()
{
    int i,j;
    int a[10];
    int iTemp;
    printf("为数组元素赋值：\n");
    /*通过键盘为数组元素赋值*/
    for(i=0;i<10;i++)
    {
        printf("a[%d]=",i);
        scanf("%d", &a[i]);
    }

    /*从小到大排序*/
    for(i=0;i<9;i++)                        /*外层循环元素下标为 0~8*/
```

```
    {
        for(j=i+1;j<10;j++)             /*内层循环元素下标为 i+1 到 9*/
        {
            if(a[j] < a[i])             /*如果当前值比其他值大*/
            {
                /*交换两个数值*/
                iTemp = a[i];
                a[i] = a[j];
                a[j] = iTemp;
            }
        }
    }

    /*输出数组*/
    for(i=0;i<10;i++)
    {
        printf("%d\t",a[i]);            /*输出制表位*/
        if(i == 4)                      /*如果是第 5 个元素*/
            printf("\n");               /*输出换行*/
    }

    return 0;                           /*程序结束*/
}
```

（1）声明一个整型数组，并通过键盘为数组元素赋值。

（2）设置一个嵌套循环，第一层循环为前 9 个数组元素，然后在第二层循环中，使用第一个数组元素分别与后面的数组元素依次进行比较，如果后面的数组元素值小于当前数组元素值，则交换两个元素值，然后使用交换后的第一个数组元素继续与后面的数组元素进行比较，直到本次循环结束。将最小的数组元素值交换到第一个数组元素的位置，然后从第二个数组元素开始，继续与后面的数组元素进行比较。依此类推，直到循环结束，就将数组元素按照从小到大的顺序重新排列了。

（3）循环输出数组中的元素，并在输出 5 个元素以后进行换行，在下一行输出后面的 5 个元素。

运行程序，显示效果如图 8.15 所示。



图 8.15 交换法排序

8.5.4 插入法排序

插入法排序较为复杂，其基本原理是抽出一个数据，在前面的数据中寻找相应的位置插入，然后继续下一个数据，直到完成排序。

下面以数字 9、6、15、4、2 为例，进行插入法排序。每次排序后的顺序如表 8.4 所示。

表 8.4　插入法排序

数组元素 / 排序过程	元素【0】	元素【1】	元素【2】	元素【3】	元素【4】
起始值	9	6	15	4	2
第 1 次	9				
第 2 次	6	9			
第 3 次	6	9	15		
第 4 次	4	6	9	15	
排序结果	2	4	6	9	15

可以发现，在第一次排序过程中将第一个数取出来，并放置在第一个位置；然后取出第二个数，并将第二个数与第一个数进行比较，如果第二个数小于第一个数，则将第二个数排在第一个数之前，否则将第二个数排在第一个数之后；然后取出下一个数，先与排在后面的数字进行比较，如果当前数字比较大则排在最后，如果当前数字比较小，还要与之前的数字进行比较，如果当前数字比前面的数字小，则将当前数字排在比它小的数字和比它大的数字之间，如果没有比当前数字小的数字，则将当前数字排在最前方；依此类推，不断取出未进行排序的数字与排序好的数字进行比较，并插入相应的位置，直到将一组数字按从小到大排序为止。

下面通过实例来看一下如何使用插入法实现数组元素从小到大的排序。

【例 8.15】　插入法排序。（**实例位置：资源包\TM\sl\8\15**）

在本实例中，声明了一个整型数组和两个整型变量，其中整型数组用于存储用户输入的数字，而两个整型变量分别作为两个元素交换时的中间变量和记录数组元素位置，然后通过双层循环进行交换法排序，最后将排序好的数组进行输出。

```
#include<stdio.h>
int main()
{
	int i;
	int a[10];
	int iTemp;
	int iPos;
	printf("为数组元素赋值：\n");
	/*通过键盘为数组元素赋值*/
	for(i=0;i<10;i++)
	{
		printf("a[%d]=",i);
		scanf("%d", &a[i]);
	}
```

```
    /*从小到大排序*/
    for(i=1;i<10;i++)                          /*循环数组中的元素*/
    {
        iTemp = a[i];                          /*设置插入值*/
        iPos = i-1;
        while((iPos>=0) && (iTemp<a[iPos]))    /*寻找插入值的位置*/
        {
            a[iPos+1] = a[iPos];               /*插入数值*/
            iPos–;
        }
        a[iPos+1] = iTemp;
    }

    /*输出数组*/
    for(i=0;i<10;i++)
    {
        printf("%d\t",a[i]);                   /*输出制表位*/
        if(i == 4)                             /*如果是第 5 个元素*/
            printf("\n");                      /*输出换行*/
    }

    return 0;                                  /*程序结束*/
}
```

（1）声明一个整型数组，并通过键盘为数组元素赋值。

（2）设置一个嵌套循环，第一层循环为后 9 个数组元素，将第二个元素赋值给中间变量，并记录前一个数组元素的下标位置。在第二层循环中，首先要判断是否符合循环的条件，允许循环的条件是记录的下标位置必须大于等于第一个数组元素的下标位置，并且中间变量的值小于之前设置下标位置的数组元素，如果满足循环条件，则将设置下标位置的数组元素值赋值给当前的数组元素，然后将记录的数组元素下标位置向前移动一位，继续进行循环判断。内层循环结束以后，将中间变量中保存的数值赋值给当前记录的下标位置之后的数组元素，继续进行外层循环，将数组中下一个数组元素赋值给中间变量，再通过内层循环进行排序，依此类推，直到循环结束，就将数组元素按照从小到大的顺序重新排列了。

（3）循环输出数组中的元素，并在输出 5 个元素以后进行换行，在下一行输出后面的 5 个元素。

运行程序，显示效果如图 8.16 所示。



图 8.16　插入法排序

8.5.5 折半法排序

折半法排序又称为快速排序，是选择一个中间值 middle（在程序中使用数组中间值），然后把比中间值小的数据放在左边，比中间值大的数据放在右边（具体的实现是从两边找，找到一对后进行交换），然后对两边分别递归使用这个过程。

下面以数字 9、6、15、4、2 为例，对这几个数字进行折半法排序。每次排序后的顺序如表 8.5 所示。

表 8.5 折半法排序

数组元素 / 排序过程	元素【0】	元素【1】	元素【2】	元素【3】	元素【4】
起始值	9	6	15	4	2
第 1 次	9	6	2	4	15
第 2 次	4	6	2	9	15
第 3 次	4	2	6	9	15
第 4 次	2	4	6	9	15
排序结果	2	4	6	9	15

可以发现，在第一次排序过程中，首先获取数组中间元素的值 15，从左右两侧分别取出数组元素与中间值进行比较。如果左侧取出的值比中间值小，则取下一个数组元素与中间值进行比较；如果左侧取出的值比中间值大，则交换两个互相比较的数组元素值。右侧的比较正好与左侧相反，当右侧取出的值比中间值大时，取前一个数组元素的值与中间值进行比较；如果右侧取出的值比中间值小，则交换两个互相比较的数组元素值。当中间值两侧的数据都比较一遍以后，数组以第一个元素为起点，以中间值的元素为终点，以上面的比较方法继续进行比较；而右侧以中间值的元素为起点，以数组最后一个元素为终点，以上述的方法进行比较。当比较完成以后，继续以折半的方式进行比较，直到将一组数字按从小到大排序为止。

下面通过实例来看一下如何使用折半法实现数组元素从小到大的排序。

【例 8.16】 折半法排序。（**实例位置：资源包\TM\sl\8\16**）

在本实例中，声明了一个整型数组，用于存储用户输入的数字，再定义一个函数，用户对数组元素进行排序，最后将排好序的数组进行输出。

注意

为了实现折半法排序，需要使用函数的递归，这部分内容将在第 9 章进行介绍，读者可以参考后面的内容进行学习。

```
#include<stdio.h>

/*声明函数*/
void CelerityRun(int left, int right, int array[]);
```

```
int main()
{
	int i;
	int a[10];
	printf("为数组元素赋值：\n");
	/*通过键盘为数组元素赋值*/
	for(i=0;i<10;i++)
	{
		printf("a[%d]=",i);
		scanf("%d", &a[i]);
	}

	/*从小到大排序*/
	CelerityRun(0,9,a);

	/*输出数组*/
	for(i=0;i<10;i++)
	{
		printf("%d\t",a[i]);                    /*输出制表位*/
		if(i == 4)                              /*如果是第 5 个元素*/
			printf("\n");                       /*输出换行*/
	}

	return 0;                                   /*程序结束*/
}

void CelerityRun(int left, int right, int array[])
{
	int i,j;
	int middle,iTemp;
	i = left;
	j = right;
	middle = array[(left+right)/2];             /*求中间值*/
	do
	{
		while((array[i]<middle) && (i<right))   /*从左找小于中值的数*/
			i++;
		while((array[j]>middle) && (j>left))    /*从右找大于中值的数*/
			j--;
		if(i<=j)                                /*找到了一对值*/
		{
			iTemp = array[i];
			array[i] = array[j];
			array[j] = iTemp;
			i++;
			j--;
		}
	}while(i<=j);                               /*如果两边的下标交错，就停止（完成一次）*/

	/*递归左半边*/
```

```
    if(left<j)
        CelerityRun(left,j,array);
    /*递归右半边*/
    if(right>i)
        CelerityRun(i,right,array);
}
```

（1）声明一个整型数组，并通过键盘为数组元素赋值。

（2）定义一个函数，用于对数组元素进行排序，函数的 3 个参数分别表示递归调用时数组最开始的元素、最后元素的下标位置以及要排序的数组。声明两个整型变量，作为控制排序算法循环的条件，分别将两个参数赋值给变量 i 和 j，i 表示左侧下标，j 表示右侧下标。首先使用 do...while 语句设计外层循环，条件为 i 小于 j，表示如果两边的下标交错就停止循环，内层的两个循环分别用来比较中间值两侧的数组元素，当左侧的数值小于中间值时，取下一个元素与中间值进行比较，否则退出第一个内层循环；当右侧的数值大于中间值时，取前一个元素与中间值进行比较，否则退出第二个内层循环。然后判断 i 的值是否小于等于 j，如果是，则交换以 i 和 j 为下标的两个元素值，继续进行外层循环。当外层循环结束以后，以数组第一个元素到以 j 为下标的元素为参数递归调用该函数，同时，以 i 为下标的数组元素到数组最后一个参数也作为参数递归调用该函数。依此类推，直到将数组元素按照从小到大的顺序重新排列为止。

（3）循环输出数组中的元素，并在输出 5 个元素以后进行换行，在下一行输出后面的 5 个元素。

运行程序，显示效果如图 8.17 所示。



图 8.17　折半法排序

8.5.6　排序算法的比较

前面介绍了 5 种排序方法，在具体应用时应该使用哪种方法呢？此时应该根据需要进行选择。下面对这 5 种排序方法进行一下简单的比较。

1．选择法排序

选择法排序在排序过程中共需要进行 n(n−1)/2 次比较，互相交换 n−1 次。选择法排序简单、容易实现，适用于数量较小的排序。

2．冒泡法排序

最好的情况是正序，只要比较一次即可；最坏的情况是逆序，需要比较 n^2 次。冒泡法排序是相对

稳定的排序方法，当待排序列有序时，效果比较好。

3．交换法排序

交换法排序和冒泡法排序类似，正序时最快，逆序时最慢，排列有序数据时效果最好。

4．插入法排序

此算法需要经过 n-1 次插入过程，如果数据恰好应该插入序列的最后端，则不需要移动数据，可节省时间。因此，若原始数据基本有序，此算法具有较快的运算速度。

5．折半法排序

对于较大的 n，折半法排序是速度最快的排序算法。但当 n 很小时，此方法往往比其他排序算法还要慢。折半法排序是不稳定的，对应有相同关键字的记录，排序后的结果可能会颠倒次序。

插入法、冒泡法、交换法排序的速度较慢，但当参加排序的序列局部或整体有序时，这种排序能达到较快的速度；在这种情况下，折半法排序反而会显得速度慢了。当 n 较小，对稳定性不作要求时，宜选用选择法排序；对稳定性有要求时，宜选用插入法或冒泡法排序。



8.6　字符串处理函数

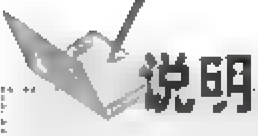
在编写程序时，经常需要对字符和字符串进行操作，如转换字符的大小写、求字符串长度等，这些都可以使用字符函数和字符串函数来解决。C 语言标准函数库专门为其提供了一系列处理函数。在编写程序的过程中，合理、有效地使用这些字符串函数，可以提高编程效率，同时也可以提高程序性能。本节将常用的对字符串处理函数进行介绍。

8.6.1　字符串复制

在字符串操作中，字符串复制是比较常用的操作之一。在字符串处理函数中包含 strcpy 函数，该函数可用于复制特定长度的字符串到另一个字符串中。其语法格式如下：

```
strcpy(目的字符数组名,源字符数组名)
```

功能：把源字符数组中的字符串复制到目的字符数组中，字符串结束标志“\0”也一同复制。


说明

（1）要求目的字符数组有足够的长度，否则不能全部装入所复制的字符串。

（2）“目的字符数组名”必须写成数组名形式；而“源字符数组名”可以是字符数组名，也可以是一个字符串常量，这时相当于把一个字符串赋予一个字符数组。

（3）不能用赋值语句将一个字符串常量或字符数组直接赋给一个字符数组。

下面通过实例来介绍一下 strcpy 函数的使用。

【例 8.17】 字符串复制。（**实例位置：资源包\TM\sl\8\17**）

本实例中，在 main 函数体中定义了两个字符数组，分别用于存储源字符数组和目的字符数组，然后获取用户为两个字符数组赋值的字符串，并分别输出两个字符数组，调用 strcpy 函数将源字符数组中的字符串赋值给目的字符数组，最后输出目的字符数组。

```
#include<stdio.h>
#include<string.h>

int main()
{
    char str1[30],str2[30];
    printf("输入目的字符串：\n");
    gets(str1);                                 /*输入目的字符串*/
    printf("输入源字符串：\n");
    gets(str2);                                 /*输入源字符串*/

    printf("输出目的字符串：\n");
    puts(str1);                                 /*输出目的字符串*/
    printf("输出源字符串：\n");
    puts(str2);                                 /*输出源字符串*/
    strcpy(str1,str2);                          /*调用 strcpy 函数实现字符串复制*/
    printf("调用 strcpy 函数进行字符串复制：\n");
    printf("复制字符串之后的目的字符串：\n");
    puts(str1);                                 /*输出复制后的目的字符串*/

    return 0;                                   /*程序结束*/
}
```

运行程序，字符串复制效果如图 8.18 所示。



图 8.18　字符串复制

8.6.2　字符串连接

字符串连接就是将一个字符串连接到另一个字符串的末尾，使其组合成一个新的字符串。在字符串处理函数中，strcat 函数就具有字符串连接的功能。其语法格式如下：

```
strcat(目的字符数组名,源字符数组名)
```

功能：把源字符数组中的字符串连接到目的字符数组中字符串的后面，并删去目的字符数组中原

有的串结束标志“\0”。



说明

目的字符数组应有足够的长度，否则很可能无法装下连接后的字符串。

下面通过实例介绍一下 strcat 函数的使用。

【例 8.18】　字符串连接。（**实例位置：资源包\TM\sl\8\18**）

在本实例的 main 函数体中定义两个字符数组，分别为存储源字符数组和目的字符数组，然后获取用户为两个字符数组赋值的字符串，并分别输出两个字符数组，调用 strcat 函数将源字符数组中的字符串连接到目的字符数组中字符串的后面，最后输出目的字符数组。

```
#include<stdio.h>
#include<string.h>

int main()
{
    char str1[30],str2[30];
    printf("输入目的字符串：\n");
    gets(str1);                                         /*输入目的字符串*/
    printf("输入源字符串：\n");
    gets(str2);                                         /*输入源字符串*/

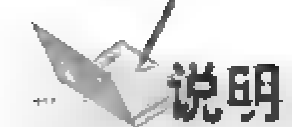
    printf("输出目的字符串：\n");
    puts(str1);                                         /*输出目的字符串*/
    printf("输出源字符串：\n");
    puts(str2);                                         /*输出源字符串*/
    strcat(str1,str2);                                  /*调用 strcat 函数进行字符串连接*/
    printf("调用 strcat 函数进行字符串连接：\n");
    printf("字符串连接之后的目的字符串：\n");
    puts(str1);                                         /*输出连接后的目的字符串*/

    return 0;                                           /*程序结束*/
}
```

运行程序，字符串连接效果如图 8.19 所示。



图 8.19　字符串连接



说明

字符串复制实质上是用源字符数组中的字符串覆盖目的字符数组中的字符串，而字符串连接则不存在覆盖的问题，只是单纯地将源字符数组中的字符串连接到目的字符数组的字符串的后面。

8.6.3 字符串比较

字符串比较就是将一个字符串与另一个字符串从首字母开始，按照 ASCII 码的顺序逐个进行比较。在字符串处理函数中，strcmp 函数就具有在字符串间进行比较的功能。其语法格式如下：

```
strcmp(字符数组名 1,字符数组名 2)
```

功能：按照 ASCII 码顺序比较两个数组中的字符串，并返回比较结果。

返回值如下：

☑ 字符串 1=字符串 2，返回值为 0。
☑ 字符串 1>字符串 2，返回值为正数。
☑ 字符串 1<字符串 2，返回值为负数。

说明

当两个字符串进行比较时，若出现不同的字符，则以第一个不同字符的比较结果作为整个比较的结果。

下面通过实例介绍一下 strcmp 函数的使用。

【例 8.19】 字符串比较。（**实例位置：资源包\TM\sl\8\19**）

在本实例的 main 函数体中定义 4 个字符数组，分别用来存储用户名、密码、用户输入的用户名及密码字符串，然后调用 strcmp 函数比较用户输入的用户名和密码是否正确。

```
#include<stdio.h>
#include<string.h>

int main()
{
    char user[20] = {"mrsoft"};                    /*设置用户名字符串*/
    char password[20] = {"mrkj"};                  /*设置密码字符串*/
    char ustr[20],pwstr[20];
    int i=0;

    while(i < 3)
    {
        printf("输入用户名字符串：\n");
        gets(ustr);                                /*输入用户名字符串*/
        printf("输入密码字符串：\n");
        gets(pwstr);                               /*输入密码字符串*/
        if(strcmp(user,ustr))                      /*如果用户名字符串不相等*/
```

```
        {
            printf("用户名字符串输入错误！\n");                  /*提示用户名字符串输入错误*/
        }
        else                                                     /*用户名字符串相等*/
        {
            if(strcmp(password,pwstr))                           /*如果密码字符串不相等*/
            {
                printf("密码字符串输入错误！\n");                /*提示密码字符串输入错误*/
            }
            else                                                 /*用户名和密码字符串都正确*/
            {
                printf("欢迎使用！\n");                          /*输出欢迎字符串*/
                break;
            }
        }
        i++;
    }
    if(i == 3)
    {
        printf("输入字符串错误 3 次！\n");                       /*输入字符串错误 3 次*/
    }

    return 0;                                                    /*程序结束*/
}
```

运行程序，字符串比较效果如图 8.20 所示。



图 8.20　字符串比较

8.6.4　字符串大小写转换

字符串的大小写转换需要使用 strupr 和 strlwr 函数。strupr 函数的语法格式如下：

```
strupr(字符串)
```

功能：将字符串中的小写字母转换成大写字母，其他字母不变。

strlwr 函数的语法格式如下：

```
strlwr(字符串)
```

功能：将字符串中的大写字母转换成小写字母，其他字母不变。

下面通过实例介绍一下 strupr 和 strlwr 函数的使用。

【例 8.20】 字符串大小写转换。（**实例位置：资源包\TM\sl\8\20**）

在本实例的 main 函数体中定义两个字符数组，分别用来存储要转换的字符串和转换后的字符串，然后根据用户输入的操作指令，判断应调用 strupr 函数还是 strlwr 函数，进行大小写转换。

```
#include<stdio.h>
#include<string.h>

int main()
{
    char text[20],change[20];
    int num;
    int i=0;

    while(1)
    {

        printf("输入转换大小写方式（1 表示大写，2 表示小写，0 表示退出）：\n");
        scanf("%d", &num);
        if(num == 1)                                       /*如果是转换为大写*/
        {
            printf("输入一个字符串：\n");
            scanf("%s", &text);                            /*输入要转换的字符串*/
            strcpy(change,text);                           /*复制要转换的字符串*/
            strupr(change);                                /*字符串转换大写*/
            printf("转换成大写字母的字符串为：%s\n",change);  /*输出转换后的字符串*/
        }
        else if(num == 2)                                  /*如果是转换为小写*/
        {
            printf("输入一个字符串：\n");
            scanf("%s", &text);                            /*输入要转换的字符串*/
            strcpy(change,text);                           /*复制要转换的字符串*/
            strlwr(change);                                /*字符串转换小写*/
            printf("转换成小写字母的字符串为：%s\n",change);  /*输出转换后的字符串*/
        }
        else if(num == 0)                                  /*如果命令字符为 0*/
        {
            break;                                         /*跳出当前循环*/
        }
    }

    return 0;                                              /*程序结束*/
}
```

运行程序，字符串大小写转换效果如图 8.21 所示。

```
输入转换大小写方式（1表示大写，2表示小写，0表示退出）：
1
输入一个字符串：
mrsoft
转换成大写字母的字符串为:MRSOFT
输入转换大小写方式（1表示大写，2表示小写，0表示退出）：
2
输入一个字符串：
MRKJ
转换成小写字母的字符串为:mrkj
输入转换大小写方式（1表示大写，2表示小写，0表示退出）：
0
Press any key to continue
```

图 8.21　字符串大小写转换

8.6.5　获得字符串长度

在使用字符串时，有时需要动态获得字符串的长度，通过循环来判断字符串结束标志“\0”虽然也能获得字符串的长度，但是实现起来相对较烦琐，这时可以使用 strlen 函数来计算字符串的长度。strlen 函数的语法格式如下：

```
strlen(字符数组名)
```

功能：计算字符串的实际长度（不含字符串结束标志“\0”），函数返回值为字符串的实际长度。

下面通过实例介绍一下 strlen 函数的使用。

【例 8.21】　获得字符串长度。（**实例位置：资源包\TM\sl\8\21**）

在本实例的 main 函数体中定义两个字符数组，用来存储用户输入的字符串，然后调用 strlen 函数计算字符串长度，调用 strcat 函数将两个字符串连接在一起，并再次调用 strlen 函数计算连接后的字符串长度。

```
#include<stdio.h>
#include<string.h>

int main()
{
	char text[50],connect[50];
	int num;

	printf("输入一个字符串：\n");
	scanf("%s", &text);                          /*获取输入的字符串*/
	num = strlen(text);                          /*计算字符串长度*/
	printf("字符串的长度为：%d\n",num);           /*输出字符串长度*/
	printf("再输入一个字符串：\n");
	scanf("%s", &connect);                       /*获取输入的字符串*/
	num = strlen(connect);                       /*计算字符串长度*/
	printf("字符串的长度为：%d\n",num);           /*输出字符串长度*/
	strcat(text,connect);                        /*连接字符串*/
	printf("将两个字符串进行连接：%s\n",text);     /*输出连接后的字符串*/
	num = strlen(text);                          /*计算连接后的字符串长度*/
	printf("连接后的字符串长度为：%d\n",num);      /*输出连接后的字符串*/
```

```
    return 0;                                   /*程序结束*/
}
```

运行程序，获得字符串长度效果如图 8.22 所示。



图 8.22　获得字符串长度



8.7　数 组 应 用

“没上过战场的军人算不上真正的军人。”这句话是有一定道理的。从程序员的角度来说，只有理论而没有实际开发能力的程序员，不能够算是程序员。本节将通过 3 个数组实例，运用前面所学知识来解决开发中的一些问题，以此来巩固所学的数组知识，使大家能真正做到理论联系实战。

8.7.1　反转输出字符串

字符串操作在应用程序中经常会用到，如连接两个字符串、查找字符串等。本节需要实现的功能是反转字符串。以字符串 mrsoft 为例，其反转的结果为 tfosrm。

在程序中定义两个字符数组：一个表示源字符串，另一个表示反转后的字符串。在源字符串中从第一个字符开始遍历，读取字符数据，在目标字符串中从最后一个字符（结束标记“\0”除外）倒序遍历字符串，依次将源字符串中的第一个字符数据写入目标字符串的最后一个字符中，将源字符串中的第二个字符数据写入目标字符串的倒数第二个字符中，依此类推，这样就实现了字符串的反转。图 8.23 描述了算法的实现过程。

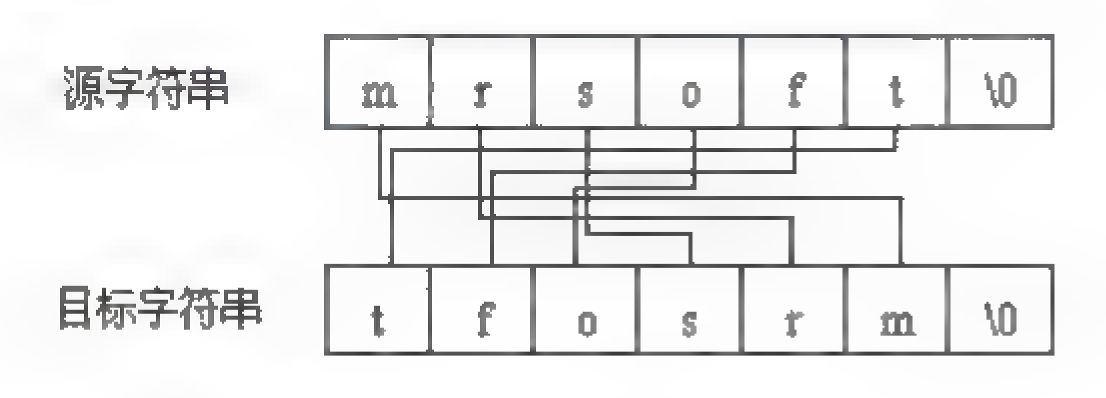


图 8.23　字符串反转示意图

下面介绍实例的设计过程。

【例 8.22】　反转输出字符串。（**实例位置：资源包\TM\sl\8\22**）

在本实例的 main 函数体中定义两个字符数组，分别为源字符数组和目标字符数组，然后在循环遍

历源字符数组的同时，将读取的字符从目标字符数组的末尾开始向前插入，最后分别输出源字符数组和目标字符数组。

```
#include<stdio.h>

int main()
{
	int i;
	char String[7]   = {"mrsoft"};
	char Reverse[7] = {0};
	int size;
	size = sizeof(String);				/*计算源字符串长度*/

	/*循环读取字符*/
	for(i=0;i<6;i++)
	{
		Reverse[size-i-2] = String[i];		/*向目标字符串插入字符*/
	}

	/*输出源字符串*/
	printf("输出源字符串：%s\n",String);
	/*输出目标字符串*/
	printf("输出目标字符串：%s\n",Reverse);

	return 0;					/*程序结束*/
}
```

运行程序，反转输出字符串的效果如图 8.24 所示。

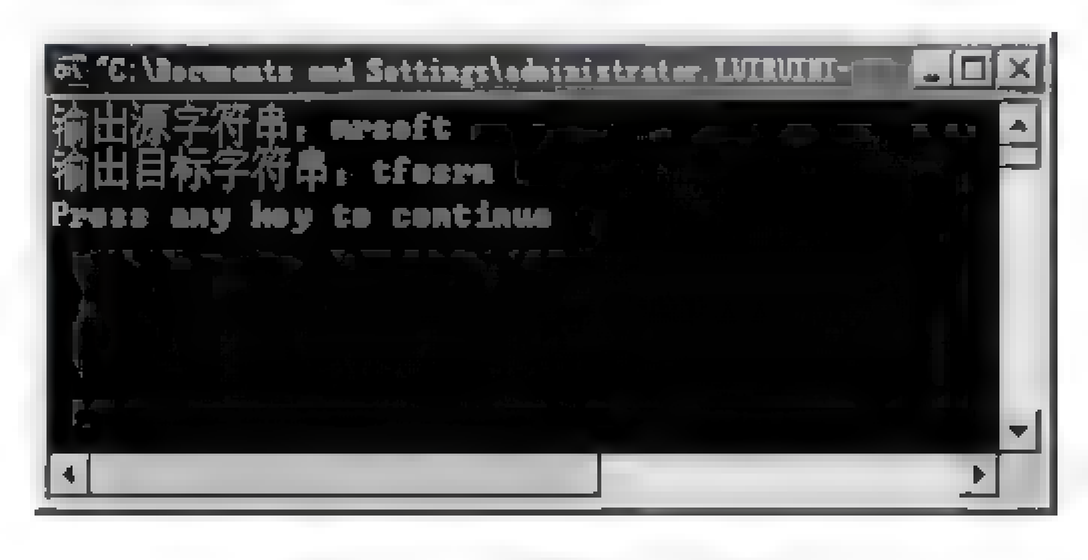


图 8.24 反转输出字符串

8.7.2 输出系统日期和时间

在控制台应用程序中，通常需要按照系统的提示信息进行操作。例如，用户进行某一个操作，需要输入一个命令，如果命令输入错误，系统会进行提示。本节要求设计一个应用程序，当用户输入命令字符“0”时显示帮助信息，输入命令字符“1”时显示系统日期，输入命令字符“2”时显示系统时间，输入命令字符“3”时退出系统。

在设计本实例时，需要解决两个问题：一是需要不断地保持程序运行，等待用户输入命令，防止 main 函数结束；二是需要获取系统日期和时间。

对于第一个问题，可以使用一个无限循环语句来实现，在循环语句中等待用户输入，如果用户输入的是命令字符“3”，则终止循环，结束应用程序。

对于第二个问题，可以使用时间函数 time 和 localtime 来获取系统的日期和时间。

下面介绍实例的实现过程。

【例 8.23】 输出系统日期和时间。（**实例位置：资源包\TM\sl\8\23**）

在本实例的 main 函数中将各个控制命令保存在数组中，然后使用 while 语句设计一个无限循环，在该循环中让用户输入命令，并判断用户输入的命令是否和数组中存储的命令相同，如果相同，则执行相应的语句。

```
#include<stdio.h>
#include<time.h>

int main()
{
    int command[4] = {0,1,2,3};                             /*定义一个数组*/
    int num;
    struct tm *sysTime;
    printf("如需帮助可输入数字 0！\n");                     /*输出字符串*/
    printf("请输入命令符：\n");                             /*输出字符串*/

    while (1)
    {
        scanf("%d", &num);                                  /*获得输入数字*/
        /*判断用于输入的字符*/
        if(command[0] == num)                               /*如果输入数字 0*/
        {
            /*输出帮助信息*/
            printf("输入数字 1 显示系统日期，输入数字 2 显示系统时间，输入数字 3 退出系统！\n");
        }
        else if(command[1] == num)                          /*如果是命令数字 1*/
        {
            time_t nowTime;
            time(&nowTime);                                 /*获取系统日期*/
            sysTime= localtime(&nowTime);                   /*转换为系统日期*/
            pnntf("系统日期：%d-%d-%d \n",1900 + sysTime->tm_year,sysTime->tm_mon + 1,sysTime->
                    tm_mday);                               /*输出信息*/
        }
        else if(command[2] == num)                          /*如果是命令数字 2*/
        {
            time_t nowTime;
            time(&nowTime);                                 /*获取系统时间*/
            sysTime = localtime(&nowTime);                  /*转换为系统时间*/
            printf("系统时间：%d:%d:%d \n",sysTime->tm_hour ,sysTime->tm_min ,sysTime-> tm_sec);
                                                            /*输出信息*/
        }
        else if(command[3] == num)
        {
```

```
            return 0;                                   /*退出系统*/
        }
        printf("请输入命令符：\n");                      /*输出字符串*/
    }

    return 0;                                           /*程序结束*/
}
```

运行程序，输出系统日期和时间的实例效果如图 8.25 所示。



图 8.25　输出系统日期和时间

8.7.3　字符串的加密和解密

在设计应用程序时，为了防止一些敏感信息泄漏，通常需要对这些信息进行加密。以用户的登录密码为例，如果密码以明文的形式存储在数据表中，就很容易被人发现；相反，如果密码以密文的形式存储，即使别人从数据表中发现了密码，这也是加密之后的密码，根本不能够使用。通过对密码进行加密，能够极大地提高系统的安全性。

为了减小本节实例的规模，这里要求设计一个加密和解密的算法，在对一个指定的字符串加密之后，利用解密函数能够对密文解密，显示明文信息。加密的方式是将字符串中每个字符加上它在字符串中的位置和一个偏移值 5。以字符串“mrsoft”为例，第一个字符 m 在字符串中的位置为 0，那么它对应的密文是'm'+0+5，即 r。

下面介绍实例的设计过程。

【例 8.24】　字符串的加密和解密。（**实例位置：资源包\TM\sl\8\24**）

在本实例的 main 函数中，使用 while 语句设计一个无限循环，并声明两个字符数组，用来保存明文和密文字符串。在首次循环中，要求用户输入字符串，进行将明文加密成密文的操作，之后的操作则是根据用户输入的命令字符进行判断：输入“1”加密新的明文，输入“2”对刚加密的密文进行解密，输入“3”退出系统。

```
#include<stdio.h>
#include<string.h>

int main()
```

```
{
    int result = 1;
    int i;
    int count = 0;
    char Text[128] = {'\0'};                        /*定义一个明文字符数组*/
    char cryptograph[128] = {'\0'};                 /*定义一个密文字符数组*/
    while (1)
    {
        if(result == 1)                             /*如果是加密明文*/
        {
            printf("请输入要加密的明文：\n");         /*输出字符串*/
            scanf("%s", &Text);                     /*获取输入的明文*/
            count = strlen(Text);
            for(i=0; i<count; i++)                  /*遍历明文*/
            {
                cryptograph[i] = Text[i] + i + 5;   /*设置加密字符*/
            }
            cryptograph[i] = '\0';                  /*设置字符串结束标记*/
            /*输出密文信息*/
            printf("加密后的密文是：%s\n",cryptograph);
        }
        else if(result == 2)                        /*如果是解密字符串*/
        {
            count = strlen(Text);
            for(i=0; i<count; i++)                  /*遍历密文字符串*/
            {
                Text[i] = cryptograph[i] - i - 5;   /*设置解密字符*/
            }
            Text[i] = '\0';                         /*设置字符串结束标记*/
            /*输出明文信息*/
            printf("解密后的明文是：%s\n",Text);
        }
        else if(result == 3)                        /*如果是退出系统*/
        {
            break;                                  /*跳出循环*/
        }
        else
        {
            printf("请输入命令符：\n");               /*输出字符串*/
        }

        /*输出字符串*/
        printf("输入 1 加密新的明文，输入 2 对刚加密的密文进行解密，输入 3 退出系统：\n");
        printf("请输入命令符：\n");                   /*输出字符串*/
        scanf("%d", &result);                       /*获取输入的命令字符*/
    }

    return 0;                                       /*程序结束*/
}
```

运行程序，字符串的加密和解密效果如图 8.26 所示。

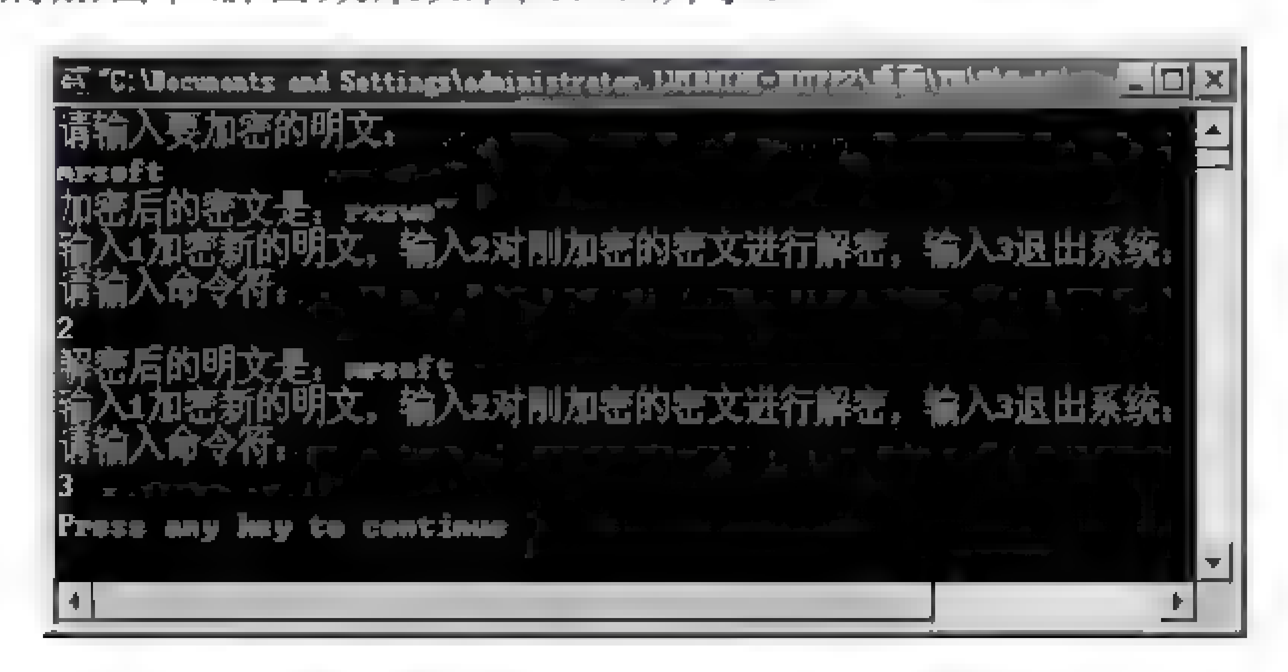


图 8.26　字符串的加密和解密

8.8　小　　结

数组类型是构造类型的一种，数组中的每个元素都属于同一种类型。本章首先介绍了有关一维数组、二维数组、字符数组及多维数组的定义和引用，使读者可以对数组有个充分的认识，然后通过实例介绍了 C 语言标准函数库中常用的字符串处理函数的使用，最后通过几个综合性的数组应用实例加深对数组的理解。

8.9　实践与练习

1．不使用 C 语言标准函数库中的函数，实现字符串的复制，即实现 strcpy 函数的功能。（**答案位置：资源包\TM\sl\8\25**）

2．某班级学习委员整理获得奖学金同学的排名，总成绩=智育成绩*60%+德育成绩*30%+体育成绩*10%，将班级前 12 名同学的成绩进行排名。（**答案位置：资源包\TM\sl\8\26**）

第 9 章

函数

（视频讲解：55分钟）

较大的程序一般应分为若干个程序模块，每个模块实现一个特定的功能。所有的高级语言中都有子程序，用来实现模块的功能。在C语言中，子程序的作用是由函数完成的。

本章致力于使读者了解关于函数的概念，掌握函数的定义及其组成部分；熟悉函数的调用方式；了解内部函数和外部函数的作用范围，区分局部变量和全局变量的不同；最后能将函数应用于程序中，将程序分成模块。

通过阅读本章，您可以：

- 了解函数的概念
- 掌握函数的定义方式
- 熟悉返回语句和函数参数的作用
- 掌握函数的调用
- 了解内部函数和外部函数的概念
- 区分局部变量和全局变量



9.1　函数概述

构成 C 程序的基本单元是函数。函数中包含程序的可执行代码。

每个 C 程序的入口和出口都位于 main 函数之中。编写程序时，并不是将所有内容都放在主函数 main 中。为了方便规划、组织、编写和调试，一般的做法是将一个程序划分成若干个程序模块，每一个程序模块都完成一部分功能。这样，不同的程序模块可以由不同的人来完成，从而可以提高软件开发的效率。

也就是说，主函数可以调用其他函数，其他函数也可以相互调用。在 main 函数中调用其他函数，这些函数执行完毕之后又返回到 main 函数中。通常把这些被调用的函数称为下层函数。函数调用发生时，立即执行被调用的函数，而调用者则进入等待的状态，直到被调用函数执行完毕。函数可以有参数和返回值。

例如，盖一栋楼房，在这项工程中，在工程师的指挥下，有工人搬运盖楼的材料，有建筑工人建造楼房，还有工人在楼房外粉刷涂料。编写程序与盖楼的道理是一样的，主函数就像工程师一样，其功能是控制每一步程序的执行，其中定义的其他函数就好比盖楼中的每一道步骤，分别去完成自己特殊的功能。

图 9.1 是某程序的函数调用示意图。

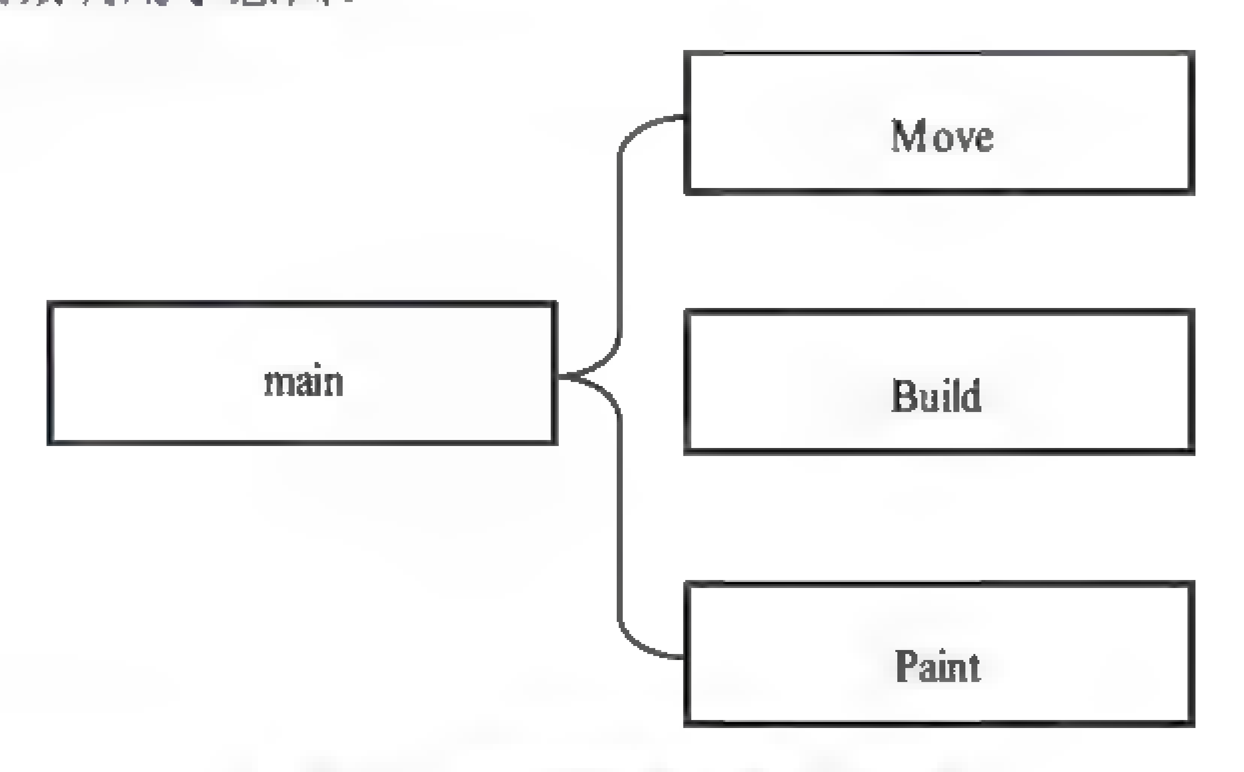


图 9.1　某程序的函数调用示意图

【例 9.1】　在主函数中调用其他函数。（**实例位置：资源包\TM\sl\9\1**）

在本实例中，通过定义函数来完成某种特定的功能。为了表示函数完成的功能，在这里使用输出的信息进行表示。希望读者通过这个实例对函数的概念有一个更为具体的认识。

```
#include<stdio.h>

void Move();                                  /*声明搬运函数*/
void Build();                                 /*声明建造函数*/
void Paint();                                 /*声明粉刷函数*/

int main()
{
    Move();                                   /*执行搬运函数*/
    Build();                                  /*执行建造函数*/
```

```
    Paint();                                    /*执行粉刷函数*/

    return 0;                                   /*程序结束*/
}

/*//////////////////////////////////////////////////////////////////////////*/
/*                          执行搬运功能                                    */
/*//////////////////////////////////////////////////////////////////////////*/
void Move()
{
    printf("This Function can move material\n");
}
/*//////////////////////////////////////////////////////////////////////////*/
/*                          执行建造功能                                    */
/*//////////////////////////////////////////////////////////////////////////*/
void Build()
{
    printf("This Function can build a building\n");
}
/*//////////////////////////////////////////////////////////////////////////*/
/*                          执行粉刷功能                                    */
/*//////////////////////////////////////////////////////////////////////////*/
void Paint()
{
    printf("This Function can paint cloth\n");
}
```

在查看程序的结果之前，先对程序进行一下分析和讲解。

☑ 首先，一个源文件由一个或者多个函数组成。一个源程序文件是一个编译单位，即以源程序为单位进行编译，而不是以函数为单位进行编译。

☑ 库函数由 C 语言系统提供，用户无须定义，在调用函数之前也不必在程序中做类型说明，只需在程序前包含有该函数原型的头文件，即可在程序中直接调用。例如，在上面程序中用于在控制台显示信息的 printf 函数，之前应在程序开始部分包含 stdio.h 这个头文件；又如要使用其他字符串操作函数 strlen、strcmp 等时，也应在程序开始部分包含 string.h。

☑ 用户自定义函数，就是用户自己编写的用来实现特定功能的函数。例如，上面程序中的 Move、Build 和 Paint 函数都是自定义函数。

☑ 在这个程序中，要使用 printf 函数首先要包含 stdio.h 头文件，然后声明 3 个自定义的函数。最后在主函数 main 中调用这 3 个函数，在主函数 main 外可以看到这 3 个函数的定义。

运行程序，显示效果如图 9.2 所示。



图 9.2　在主函数中调用其他函数



9.2 函数的定义

在程序中编写函数时，函数的定义是让编译器知道函数的功能。定义的函数包括函数头和函数体两部分。

1. 函数头

函数头分为以下 3 个部分：

☑ 返回值类型。返回值可以是某个 C 数据类型。

☑ 函数名。函数名也就是函数的标识符，函数名在程序中必须是唯一的。因为是标识符，所以函数名也要遵守标识符命名规则。

☑ 参数表。参数表可以没有变量也可以有多个变量，在进行函数调用时，实际参数将被复制到这些变量中。

2. 函数体

函数体包括局部变量的声明和函数的可执行代码。

前面最常提到的就是 main 函数，下面对其进行介绍。

所有的 C 程序都必须有一个 main 函数。该函数已经由系统声明过了，在程序中只需要定义一下即可。main 函数的返回值为整型，可以有两个参数，一个是整数，一个是指向字符数组的指针。虽然在调用时有参数传递给 main 函数，但是在定义 main 函数时可以不带任何参数，在前面的所有实例中都可以看到，main 函数就没有带任何参数。除了 main 函数外，其他函数在定义和调用时，参数都必须是匹配的。

程序中从来不会调用 main 函数，只会在开始运行程序时调用 main 函数。当 main 函数结束返回时，系统的结束过程将接收这个返回值。至于启动和结束的过程，程序员不必关心，编译器在编译和链接时会自动提供。不过根据习惯，当程序结束时，应该返回整数值。其他返回值的意义由程序的要求所决定，通常都表示程序非正常终止。

为了让读者习惯 main 函数的返回值，可以看到本书所有实例中的 main 函数都定义为如下形式：

```
int main()
{
    ...                                     /*程序代码*/
    return 0;                               /*程序结束*/
}
```

9.2.1 函数定义的形式

C 语言的库函数在编写程序时是可以直接调用的，如 printf 输出函数。而自定义函数则必须由用户对其进行定义，在其函数的定义中完成函数特定的功能，这样才能被其他函数调用。

一个函数的定义分为函数头和函数体两个部分。函数定义的语法格式如下：

```
返回值类型  函数名(参数列表)
{
    函数体(函数实现特定功能的过程);
}
```

定义一个函数的代码如下：

```
int AddTwoNumber(int iNum1,int iNum2)          /*函数头部分*/
{
    /*函数体部分，实现函数的功能*/
    int result;                                /*定义整型变量*/
    result = iNum1+iNum2;                      /*进行加法操作*/
    return result;                             /*返回操作结果，结束*/
}
```

通过代码分析一下定义函数的过程。

1．函数头

函数头用来标志一个函数代码的开始，这是一个函数的入口处。函数头分成返回值类型、函数名和参数列表 3 个部分。

在上面的代码中，函数头如图 9.3 所示。

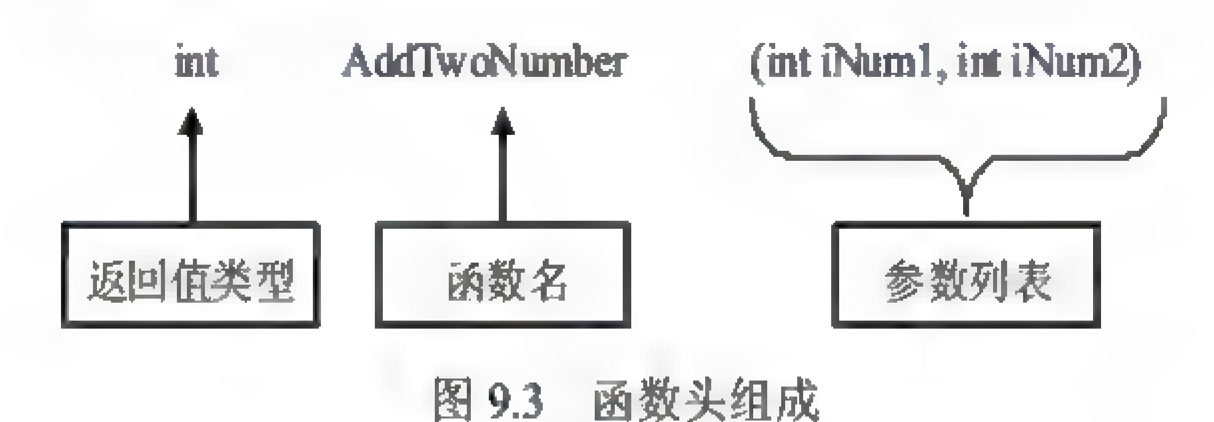


图 9.3　函数头组成

2．函数体

函数体位于函数头的下方位置，由一对大括号括起来，大括号决定了函数体的范围。函数要实现的特定功能，都是在函数体部分通过代码语句完成的，最后通过 return 语句返回实现的结果。在上面的代码中，AddTwoNumber 函数的功能是实现两个整数加法，因此定义一个整数用来保存加法的计算结果，之后利用传递进来的参数进行加法操作，并将结果保存在 result 变量中，最后函数要将所得到的结果进行返回。通过这些语句的操作，实现了函数的特定功能。

现在已经了解到定义一个函数应该使用怎样的语法格式，在定义函数时会有如下几种特殊的情况：

☑ 无参函数

无参函数也就是没有参数的函数。无参函数的语法格式如下：

```
返回值类型 函数名()
{
    函数体
}
```

通过代码来看一下无参函数。例如，使用上面的语法定义一个无参函数，代码如下：

```
void ShowTime()                                /*函数头*/
{
```

```
    printf("It's time to show yourself!");           /*显示一条信息*/
}
```

☑ 空函数

顾名思义，空函数就是没有任何内容的函数，也没有什么实际作用。空函数既然没有什么实际功能，那么为什么要存在呢？原因是空函数所处的位置是要放一个函数的，只是这个函数现在还未编好，用这个空函数先占一个位置，以后用一个编好的函数来取代它。

空函数的形式如下：

```
类型说明符 函数名()
{
}
```

例如，定义一个空函数，留出一个位置以后再添加其中的功能：

```
void ShowTime()                                   /*函数头*/
{
}
```

9.2.2　定义与声明

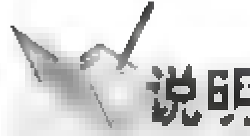
在程序中编写函数时，要先对函数进行声明，再对函数进行定义。函数的声明是让编译器知道函数的名称、参数、返回值类型等信息。函数的定义是让编译器知道函数的功能。

函数声明的格式由函数返回值类型、函数名、参数列表和分号 4 部分组成，其形式如下：

```
返回值类型　函数名(参数列表);
```

此处要注意的是，在声明的最后要用分号“;”作为语句的结尾。例如，声明一个函数的代码如下：

```
Int ShowNumber(int iNumber);
```



说明

为了使读者更容易区分函数的声明和定义，通过一个比喻来说明两者之间的关系。在生活中经常能看到很多电器的宣传广告。通过宣传广告，可以了解到电器的名称和用处等。当顾客了解这个电器之后，就会到商店里看一看这个电器，经过服务人员的介绍，就会知道电器的具体功能和使用的方式。函数的声明就相当于电器商品的宣传广告，可以帮助顾客了解电器。函数的定义就相当于服务人员具体介绍电器的功能和使用方式。

【例 9.2】　函数的定义与声明。（**实例位置：资源包\TM\sl\9\2**）

通过本实例的代码了解函数声明与函数定义的位置，及其在程序中的作用。

```
#include<stdio.h>

/*函数的声明*/
void ShowNumber(int iNumber);
```

```
int main()
{
    int iShowNumber;                                         /*定义整型变量*/
    printf("What Number do you wanna show?\n");              /*输出提示信息*/
    scanf("%d",&iShowNumber);                                /*输入整数*/
    ShowNumber(iShowNumber);                                 /*调用函数*/
    return 0;                                                /*程序结束*/
}

/*函数的定义*/
void ShowNumber(int iNumber)
{
    printf("You wanna to show the Number is:%d\n",iNumber);  /*输出整数*/
}
```

（1）观察上面的程序，可以看到在 main 函数的开头进行了 ShowNumber 函数的声明，声明的作用是告知其函数将在后面进行定义。

（2）在 main 函数体中，首先定义一个整型的变量 iShowNumber，之后输出一条提示消息。

（3）在消息提示下输入整型变量，最后调用 ShowNumber 函数进行输出操作，最后在 main 函数的定义之后就可以看到 ShowNumber 函数的定义。

运行程序，显示效果如图 9.4 所示。

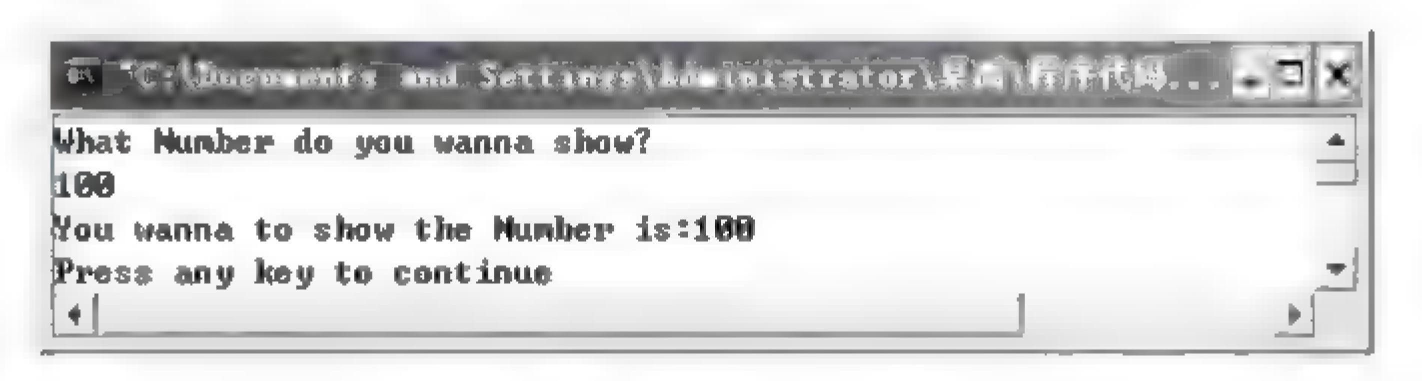


图 9.4　函数的定义与声明

注意

如果将函数的定义放在调用函数之前，就不需要进行函数的声明，此时函数的定义就包含了函数的声明。例如，将上面的程序改为如下代码：

```
/*函数的定义*/
void ShowNumber(int iNumber)
{
    printf("You wanna to show the Number is:%d\n",iNumber);  /*输出整数*/
}

int main()
{
    int iShowNumber;                                         /*定义整型变量*/
    printf("What Number do you wanna show?\n");              /*输出提示信息*/
    scanf("%d",&iShowNumber);                                /*输入整数*/
    ShowNumber(iShowNumber);                                 /*调用函数*/
    return 0;                                                /*程序结束*/
}
```



9.3 返回语句

在函数的函数体中常会看到这样一句代码：

```
return 0;
```

这就是返回语句。返回语句有以下两个主要用途：

☑ 利用返回语句能立即从所在的函数中退出，即返回到调用的程序中去。

☑ 返回语句能返回值，即将函数值赋给调用的表达式中。当然，有些函数也可以没有返回值，例如，返回值类型为 void 的函数就没有返回值。

下面对这两个用途进行说明。

9.3.1 从函数返回

从函数返回是返回语句的第一个主要用途。在程序中，有两种方法可以终止函数的执行，并返回到调用函数的位置。第一种方法是在函数体中，从第一句一直执行到最后一句，当所有语句都执行完，程序遇到结束符号“}”后返回。

【例 9.3】 从函数返回。(**实例位置：资源包\TM\sl\9\3**)

在本实例中，通过一个简单的函数，在函数的适当位置输出提示信息，进而观察函数的返回过程。

```
#include<stdio.h>

int Function();                                              /*声明函数*/

int main()
{
    printf("this step is before the Function\n");            /*输出提示信息*/
    Function();                                              /*调用函数*/
    printf("this step is end of the Function\n");            /*输出提示信息*/
    return 0;
}

int Function()                                               /*定义函数*/
{
    printf("this step is in the Function\n");                /*输出提示信息*/
    /*函数结束*/
}
```

（1）在代码中，首先声明使用的函数，在主函数中首先输出提示信息来表示此时程序执行的位置在 main 函数中。

（2）调用 Function 函数，在该函数中通过输出的提示信息表示此时程序执行的位置在 Function 中，

由于定义的函数中只有一条语句，因此执行完这条语句之后就返回到 main 函数中。

（3）自定义的函数执行完，返回到 main 函数中，继续执行一条输出语句，并显示提示信息，表示此时自定义的函数已经执行完毕。

（4）最后调用 return 函数，程序结束。

运行程序，显示效果如图 9.5 所示。

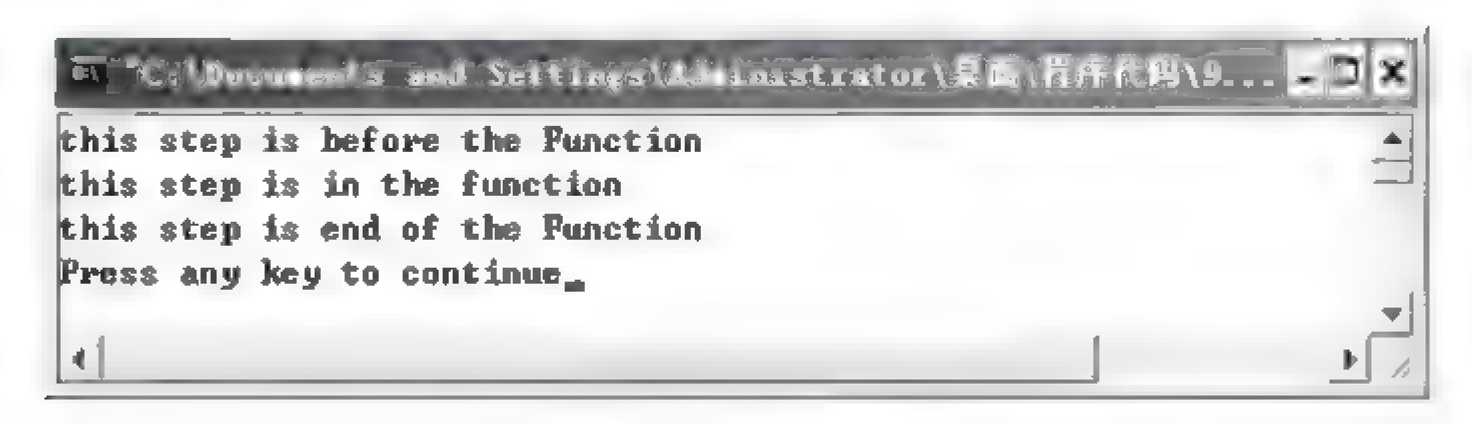


图 9.5 从函数返回

9.3.2 返回值

用户在调用某个函数时，通常是希望能得到一个确定的值，这就是函数的返回值。例如下面的代码：

```
int Minus(int iNumber1,int iNumber2)
{
    int iResult;                          /*定义一个整型变量，用来存储返回的结果*/
    iResult=iNumber1-iNumber2;            /*进行减法计算，得到计算结果*/
    return iResult;                       /*return 语句返回计算结果*/
}
int main()
{
    int iResult;                          /*定义一个整型变量*/
    iResult=Minus(9,4);                   /*进行 9-4 的减法计算，并将结果赋值给变量 iResult*/
    return 0;                             /*程序结束*/
}
```

从上面的代码中可以看到，首先定义了一个进行减法操作的函数 Minus，在 main 主函数中通过调用 Minus 函数将计算的减法结果赋值给在 main 函数中定义的变量 iResult。

下面对函数进行说明。

（1）函数的返回值都通过函数中的 return 语句获得，return 语句将被调用函数中的一个确定值返回到调用函数中。例如，上面代码中 Minus 自定义函数的最后，使用 return 语句将计算的结果返回到主函数 main 调用的位置。



return 语句后面的括号是可以省略的，例如 return 0 和 return(0)是相同的，在本书的实例中都将括号进行了省略，因此在此对 return 进行说明。

（2）函数返回值的类型。既然函数有返回值，这个值当然应该属于某一种确定的类型，因此应当在定义函数时明确指出函数返回值的类型。例如：

```
int Max(int iNum1,int iNum2);
double Min(double dNum1,double dNum2);
char Show(char cChar);
```

（3）如果函数值的类型和 return 语句中表达式的值不一致，则以函数返回值的类型为准。数值型数据可以自动进行类型转换，即函数定义的返回值类型决定最终返回值的类型。

【例 9.4】　返回值类型与 return 值类型。（**实例位置：资源包\TM\sl\9\4**）

在本实例中可以看到，自定义的函数返回值类型与最终 return 语句返回值的类型不一致，但是通过类型转换后，函数的返回类型和定义类型一致。

```
#include<stdio.h>

char ShowChar();                          /*函数的声明*/

int main()
{
    char cResult;
    cResult=ShowChar();                   /*调用 ShowChar 函数输入一个整数，并将结果赋值给变量 cResult*/
    printf("%c\n",cResult);               /*将 cResult 由原本的 int 的数值，转换为字符型，进行输出*/
    return 0;                             /*程序结束*/
}

char ShowChar()
{
    int iNumber;                          /*定义整型变量*/
    printf("please input a number:\n");   /*输出提示信息*/
    scanf("%d",&iNumber);                 /*输入一个整型变量*/
    return iNumber;                       /*返回的是整型*/
}
```

（1）在程序代码中，首先程序声明了一个 ShowChar 函数，在主函数 main 中定义一个字符型的变量 cResult，调用自定义函数 ShowChar 得到返回的值，使用 printf 函数将所得到的结果进行输出显示。

（2）在主函数 main 外定义了 ShowChar 函数，在其函数体中定义的是一个整型变量，用户通过提示信息输入数据，最后将数据进行返回。

（3）在这里可以看到，虽然在 ShowChar 函数中返回的是 int 型变量，但是由于定义时指定的返回值类型是 char 型，因此返回值是 char 类型。

运行程序，显示效果如图 9.6 所示。



图 9.6　返回值类型与 return 值类型



9.4 函数参数

在调用函数时，大多数情况下，主调函数和被调用函数之间有数据传递关系，这就是前面提到的有参数的函数形式。函数参数的作用是传递数据给函数使用，函数利用接收到的数据进行具体的操作处理。

函数参数在定义函数时放在函数名称的后面，如图 9.7 所示。

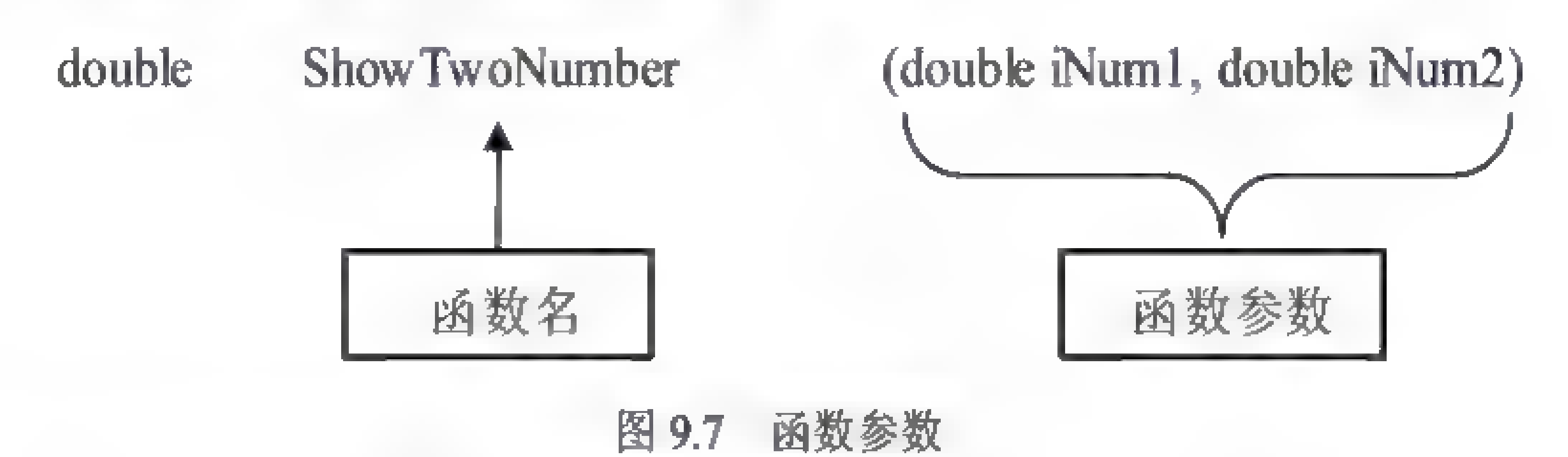


图 9.7 函数参数

9.4.1 形式参数与实际参数

在使用函数时，经常会用到形式参数和实际参数。两者都叫作参数，那么二者有什么关系？二者之间的区别是什么？两种参数分别起到什么作用？接下来通过形式参数与实际参数的名称和作用来进行理解，再通过一个比喻和实例进行深入理解。

1. 通过名称理解

☑ 形式参数：顾名思义，就是形式上存在的参数。

☑ 实际参数：即实际存在的参数。

2. 通过作用理解

☑ 形式参数：在定义函数时，函数名后面括号中的变量名称为“形式参数”。在函数调用之前，传递给函数的值将被复制到这些形式参数中。

☑ 实际参数：在调用一个函数时，也就是真正使用一个函数时，函数名后面括号中的参数为“实际参数”。函数的调用者提供给函数的参数叫实际参数。实际参数是表达式计算的结果，并且被复制给函数的形式参数。

通过图 9.8 可以更好地理解。



说明

形式参数简称为形参，实际参数简称为实参。

3. 通过一个比喻来理解形式参数和实际参数

母亲拿来了一袋牛奶，将牛奶倒入一个空奶瓶中，然后喂宝宝喝牛奶。函数的作用就相当于宝宝用奶瓶喝牛奶这个动作，实参相当于母亲拿来的一袋牛奶，而空的奶瓶就相当于形参。牛奶放入奶瓶

这个动作相当于将实参传递给形参，使用灌好牛奶的奶瓶就相当于函数使用参数进行操作的过程。

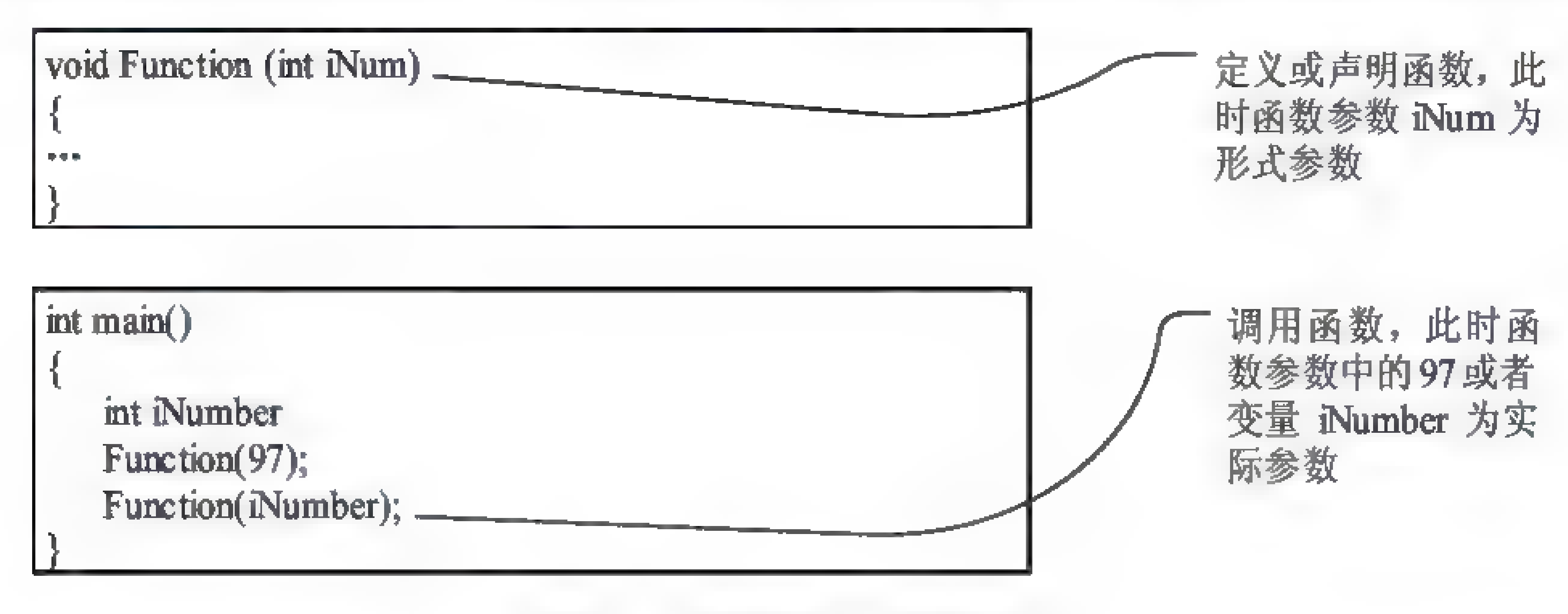


图 9.8　形式参数与实际参数

下面通过一个实例对形式参数和实际参数进行讲解。

【例 9.5】　形式参数与实际参数的比喻程序。（**实例位置：资源包\TM\sl\9\5**）

本实例中，将上面的比喻进行了模拟，希望读者可以通过动手操作，领悟上面的比喻，对形式参数和实际参数加深理解，更好地掌握知识点。

```
#include<stdio.h>

void DrinkMilk(char* cBottle);                  /*声明函数*/

int main()
{
    char cPoke[]="";                            /*定义字符数组变量*/
    printf("Mother wanna give the baby:");      /*输出信息提示*/
    scanf("%s",&cPoke);                         /*输入字符串*/
    DrinkMilk(cPoke);                           /*将实际参数传递给形式参数*/
    return 0;                                   /*程序结束*/
}

/*喝牛奶的动作*/
void DrinkMilk(char* cBottle)                   /*cBottle 为形式参数*/
{
    printf("The Baby drink the %s\n",cBottle);  /*输出提示，进行喝牛奶动作*/
}
```

现在根据上面的实例，一边理解一边对本程序进行讲解。

（1）首先声明程序中要用到的函数 DrinkMilk，在声明函数时 cBottle 变量称为形式参数，这就相当于之前母亲为孩子准备好的一袋牛奶。

（2）在主函数 main 中，定义一个字符数组变量用来保存用户输入的字符。

（3）通过 printf 库函数显示信息，表示此时孩子饿了，妈妈应该喂孩子吃东西。

（4）使用 scanf 库函数在控制台上输入字符串，将其字符串保存在 cPoke 变量中。

（5）cPoke 获得数据之后，调用 DrinkMilk 函数，将 cPoke 变量作为 DrinkMilk 函数的参数传递。

此时的 cPoke 变量就是实际参数，而传递的对象就是形式参数。这就相当于妈妈把牛奶袋打开后，将牛奶放入空奶瓶中。

（6）既然调用 DrinkMilk 函数，程序就会调转到 DrinkMilk 函数的定义处。在函数定义的函数参数 cBottle 为形式参数，不过此时 cBottle 已经得到了 cPoke 变量传递给它的值。这样，在下面使用输出语句 printf 输出 cBottle 变量时，显示的数据就是 cPoke 变量保存的数据。此时就相当于使用灌满牛奶的奶瓶喂宝宝喝牛奶一样。

（7）DrinkMilk 函数执行完，回到主函数 main 中，return 语句返回 0，程序结束。此时，宝宝已经喝饱了，妈妈就可以安心地做其他事情。

运行程序，显示效果如图 9.9 所示。



图 9.9　形式参数与实际参数的比喻程序

9.4.2　数组作函数参数

本节将讨论数组作为实参传递给函数的这种特殊情况。将数组作为函数参数进行传递，不同于标准的赋值调用的参数传递方法。

当数组作为函数的实参时，只传递数组的地址，而不是将整个数组赋值到函数中。当用数组名作为实参调用函数时，指向该数组的第一个元素的指针就被传递到函数中。

注意

C 语言中没有任何下标的数组名，而是一个指向该数组第一个元素的指针。例如，定义一个具有 10 个元素的整型数组：

```
int Count[10];                                    /*定义整型数组*/
```

其中的代码没有下标的数组名 Count 与指向第一个元素的指针*Count 是相同的。

声明函数参数时必须具有相同的类型，根据这一点，下面将对使用数组作为函数参数的各种情况进行详细的讲解。

1. 数组元素作为函数参数

由于实参可以是表达式形式，数组元素可以是表达式的组成部分，因此数组元素可以作为函数的实参，与用变量作为函数实参一样，是单向传递。

【例 9.6】　数组元素作为函数参数。（**实例位置：资源包\TM\sl\9\6**）

在实例中定义一个数组，然后将赋值后的数组元素作为函数的实参进行传递，当函数的形参得到实参传递的数值后，将其显示输出。

```
#include<stdio.h>

void ShowMember(int iMember);                    /*声明函数*/

int main()
{
    int iCount[10];                              /*定义一个整型的数组*/
    int i;                                       /*定义整型变量，用于循环*/

    for(i=0;i<10;i++)                            /*进行赋值循环*/
    {
        iCount[i]=i;                             /*为数组中的元素进行赋值操作*/
    }

    for(i=0;i<10;i++)                            /*循环操作*/
    {
        ShowMember(iCount[i]);                   /*执行输出函数操作*/
    }
    return 0;
}

void ShowMember(int iMember)                     /*函数定义*/
{
    printf("Show the member is%d\n",iMember);    /*输出数据*/
}
```

（1）在源文件的开始处为了下面要使用的函数进行声明，在主函数 main 的开始处首先定义一个整型数组和一个整型变量 i，变量 i 用于下面要使用的循环语句。

（2）变量定义完成之后要对数组中的元素进行赋值，在这里使用 for 循环语句，变量 i 作为循环语句的循环条件，并且作为数组的下标指定数组元素位置。

（3）通过一个循环语句调用 ShowMember 函数显示数据，其中可以看到 i 作为参数中数组的下标，表示指定要输出的数组元素。

运行程序，显示效果如图 9.10 所示。

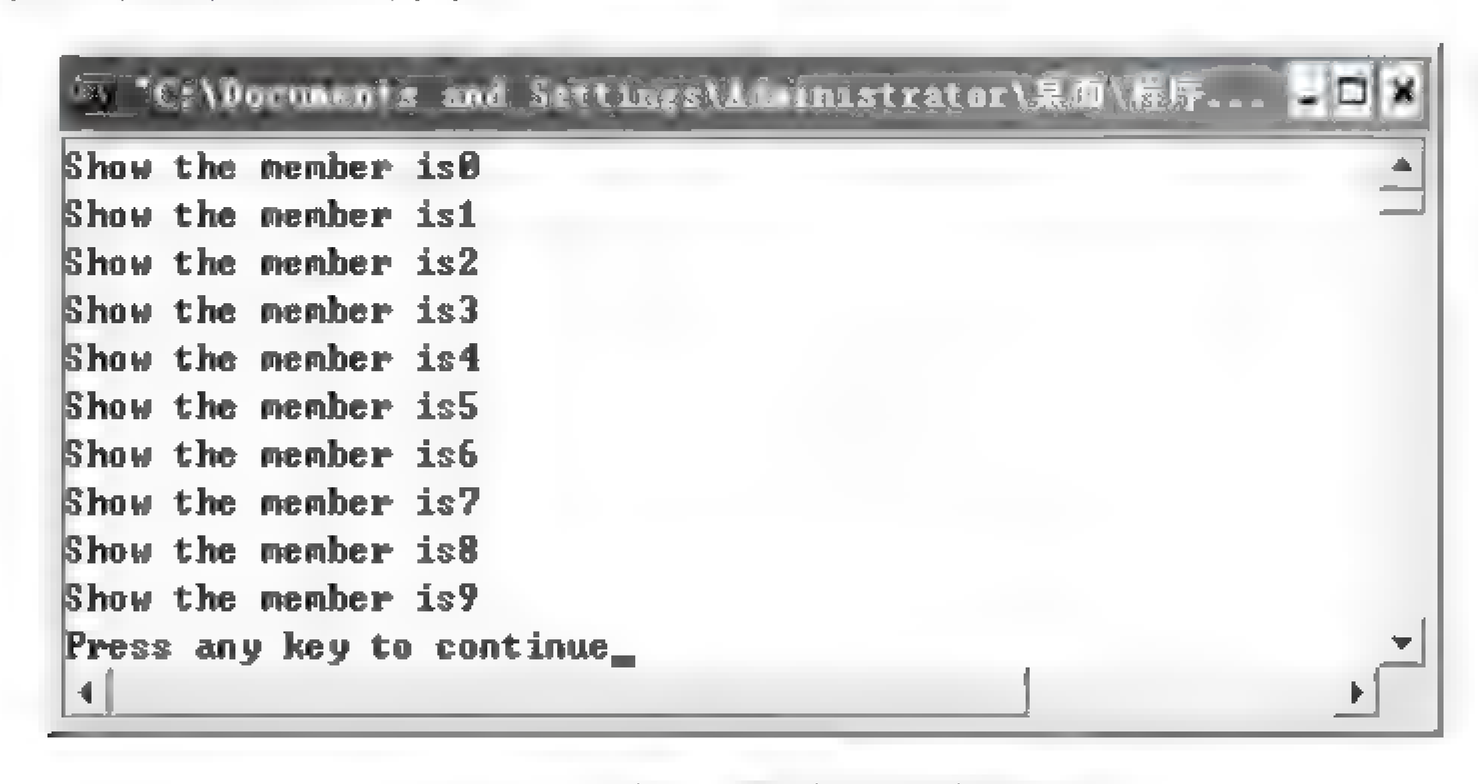


图 9.10　数组元素作为函数参数

2. 数组名作为函数参数

可以用数组名作为函数参数，此时实参与形参都应使用数组名。

【例 9.7】 数组名作为函数参数。（**实例位置：资源包\TM\sl\9\7**）

在本实例中，通过使用数组名作为函数的实参和形参，实现与例 9.6 同样的程序显示结果。

```
#include<stdio.h>

void Evaluate(int iArrayName[10]);                    /*声明赋值函数*/
void Display(int iArrayName[10]);                     /*声明显示函数*/

int main()
{
    int iArray[10];                                   /*定义一个具有 10 个元素的整型数组*/

    Evaluate(iArray[10]);                             /*调用函数进行赋值操作，将数组名作为参数*/
    Display(iArray[10]);                              /*调用函数进行赋值操作，将数组名作为参数*/
    return 0;
}
/*//////////////////////////////////////////////////////////////////////////////////////////////*/
/*                              数组元素的显示                                          */
/*//////////////////////////////////////////////////////////////////////////////////////////////*/
void Display(int iArrayName[10])
{
    int i;                                            /*定义整型变量*/
    for(i=0;i<10;i++)                                 /*执行循环语句*/
    {                                                 /*在循环语句中执行输出操作*/
        printf("the member number is %d\n",iArrayName[i]);
    }
}
/*//////////////////////////////////////////////////////////////////////////////////////////////*/
/*                              进行数组元素的赋值                                      */
/*//////////////////////////////////////////////////////////////////////////////////////////////*/
void Evaluate(int iArrayName[10])
{
    int i;                                            /*定义整型变量*/
    for(i=0;i<10;i++)                                 /*执行循环语句*/
    {                                                 /*在循环语句中执行赋值操作*/
        iArrayName[i]=i;
    }
}
```

（1）首先是对程序中将要使用的函数进行声明操作，在声明语句中可以看到函数参数中是用数组名作为参数名。

（2）在主函数 main 中，定义一个具有 10 个元素的整型数组 iArray。

（3）定义整型数组之后，调用 Evaluate 函数，这时可以看到 iArray 作为函数参数传递数组的地址。在 Evaluate 的定义中可以看到，通过使用形参 iArrayName 对数组进行了赋值操作。

（4）调用 Evaluate 函数后，整型数组已经被赋值，此时又调用 Display 函数将其数组进行输出，

可以看到在函数参数中使用的也是数组名称。

运行程序，显示效果如图 9.11 所示。

```
C:\Documents and Settings\Administrator\桌面\程序.
the member number is 0
the member number is 1
the member number is 2
the member number is 3
the member number is 4
the member number is 5
the member number is 6
the member number is 7
the member number is 8
the member number is 9
Press any key to continue
```

图 9.11　数组名作为函数参数

3. 可变长度数组作为函数参数

可以将函数的参数声明成长度可变的数组，在此基础上利用上面的程序进行修改。声明方式的代码如下：

```
void Function(iint iArrayName[]);                     /*声明函数*/

int iArray[10];                                       /*定义整型数组*/
Function(iArray);                                     /*将数组名作为实参进行传递*/
```

从上面的代码中可以看到，在定义和声明一个函数时将数组作为函数参数，并且没有指明数组此时的大小，这样就将函数参数声明为数组长度可变的数组。

【例 9.8】　可变长度数组作为函数参数。（**实例位置：资源包\TM\sl\9\8**）

在本实例中，修改例 9.7，使其参数为可变长度数组，通过比较两个程序，使读者对此加深印象。

```
#include<stdio.h>

void Evaluate(int iArrayName[]);                      /*声明函数，参数为可变长度数组*/
void Display(int iArrayName[]);                       /*声明函数，参数为可变长度数组*/

int main()
{
    int iArray[10];                                   /*定义一个具有 10 个元素的整型数组*/

    Evaluate(iArray[10]);                             /*调用函数进行赋值操作，将数组名作为参数*/
    Display(iArray[10]);                              /*调用函数进行赋值操作，将数组名作为参数*/
    return 0;
}
/*////////////////////////////////////////////////////////////////////////////////////*/
/*                          数组元素的显示                                              */
/*////////////////////////////////////////////////////////////////////////////////////*/
void Display(int iArrayName[])                        /*定义函数，参数为可变长度数组*/
{
    int i;                                            /*定义整型变量*/
    for(i=0;i<10;i++)                                 /*执行循环语句*/
```

```
    {                                                   /*在循环语句中执行输出操作*/
        printf("the member number is %d\n",iArrayName[i]);
    }
}
/*//////////////////////////////////////////////////////////////////////////*/
/*                    进行数组元素的赋值                                      */
/*//////////////////////////////////////////////////////////////////////////*/
void Evaluate(int iArrayName[])                         /*定义函数，参数为可变长度数组*/
{
    int i;                                              /*定义整型变量*/
    for(i=0;i<10;i++)                                   /*执行循环语句*/
    {                                                   /*在循环语句中执行赋值操作*/
        iArrayName[i]=i;
    }
}
```

本程序的执行过程与例 9.7 相似，只是在声明和定义函数参数时，使用的是可变长度数组的形式。运行程序，显示效果如图 9.12 所示。

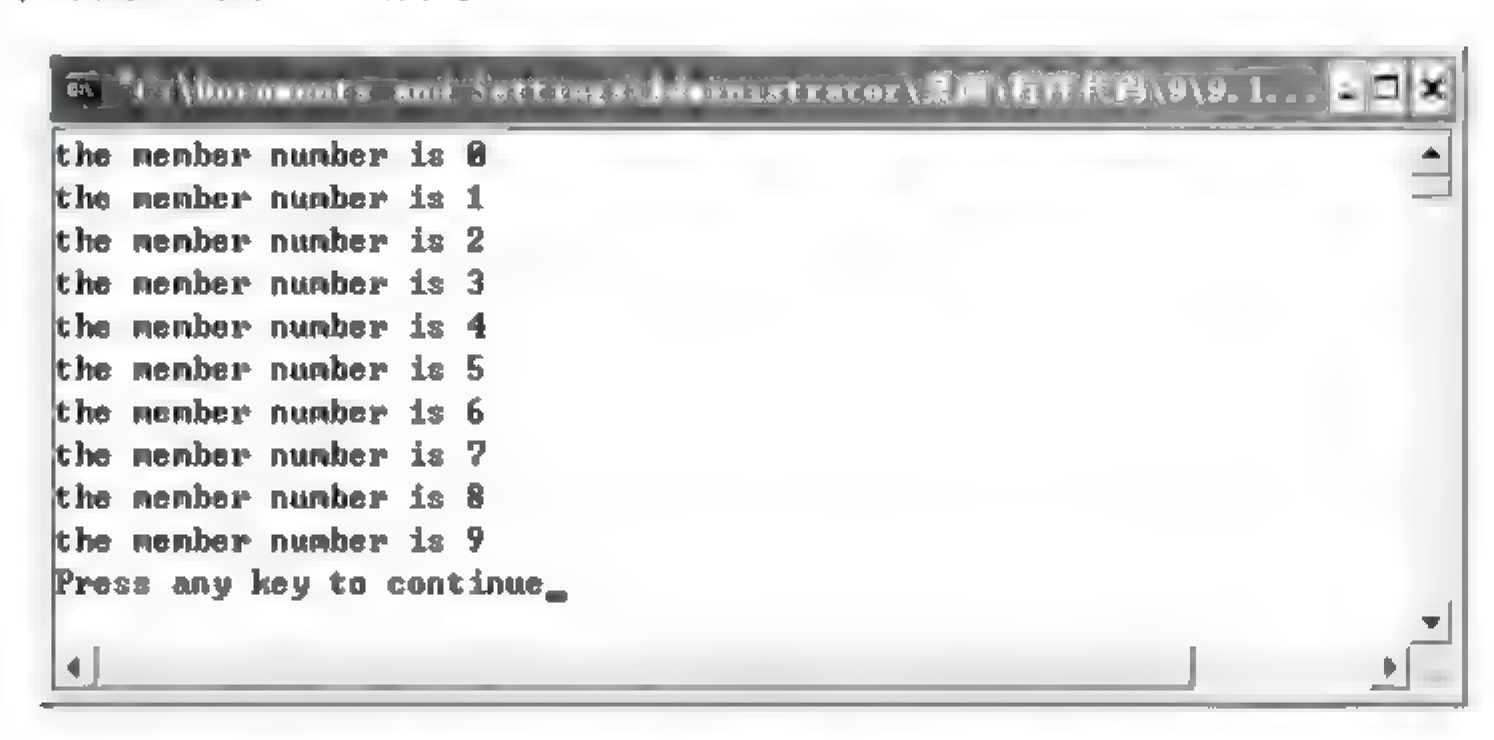


图 9.12　可变长度数组作为函数参数

4．使用指针作为函数参数

最后一种方式是将函数参数声明为一个指针。在前面的介绍中也曾提到，当数组作为函数的实参时，只传递数组的地址，而不是将整个数组赋值到函数中去。当用数组名作为实参调用函数时，指向该数组的第一个元素的指针就会被传递到函数中。



说明

将函数参数声明为一个指针，也是 C 语言程序比较常见的用法。

例如，声明一个函数参数为指针时，传递数组的方法如下：

```
void Function(int* pPoint);                 /*声明函数*/

int iArray[10];                             /*定义整型数组*/
Function(iArray);                           /*将数组名作为实参进行传递*/
```

从上面的代码中可以看到，指针在声明 Function 时作为函数参数。在调用函数时，可以将数组名作为函数的实参进行传递。

【例 9.9】　指针作为函数参数。（实例位置：资源包\TM\sl\9\9）

在本实例中，仍然实现与前面相同的功能。在之前实例程序的基础上进行修改，使之满足新的情况。

```
#include<stdio.h>

void Evaluate(int* pPoint);                    /*声明函数，参数为可变长度数组*/
void Display(int* pPoint);                     /*声明函数，参数为可变长度数组*/

int main()
{
    int iArray[10];                            /*定义一个具有 10 个元素的整型数组*/

    Evaluate(iArray);                          /*调用函数进行赋值操作，将数组名作为参数*/
    Display(iArray);                           /*调用函数进行赋值操作，将数组名作为参数*/
    return 0;
}
/*//////////////////////////////////////////////////////////////////////////////*/
/*                     数组元素的显示                                          */
/*//////////////////////////////////////////////////////////////////////////////*/
void Display(int* pPoint)                      /*定义函数，参数为可变长度数组*/
{
    int i;                                     /*定义整型变量*/
    for(i=0;i<10;i++)                          /*执行循环的语句*/
    {                                          /*在循环语句中执行输出操作*/
        printf("the member number is %d\n",pPoint[i]);
    }
}
/*//////////////////////////////////////////////////////////////////////////////*/
/*                     进行数组元素的赋值                                      */
/*//////////////////////////////////////////////////////////////////////////////*/
void Evaluate(int* pPoint)                     /*定义函数，参数为可变长度数组*/
{
    int i;                                     /*定义整型变量*/
    for(i=0;i<10;i++)                          /*执行循环语句*/
    {                                          /*在循环语句中执行赋值操作*/
        pPoint[i]=i;
    }
}
```

（1）在程序的开始处声明函数时，将指针声明为函数参数。

（2）在主函数 main 中，首先定义一个具有 10 个元素的数组。

（3）将数组名作为 Evaluate 函数的参数。在 Evaluate 函数的定义中，可以看到定义函数参数也为指针。在 Evaluate 函数体内，通过循环对数组进行赋值操作。可以看到虽然 pPoint 是指针，但也可以使用数组的形式进行表示。

（4）在主函数 main 中调用 Display 函数，进行显示输出操作。

运行程序，显示效果如图 9.13 所示。

```
the member number is 0
the member number is 1
the member number is 2
the member number is 3
the member number is 4
the member number is 5
the member number is 6
the member number is 7
the member number is 8
the member number is 9
Press any key to continue
```

图 9.13　指针作为函数参数

9.4.3　main 函数的参数

在前面介绍函数定义的内容中，曾在讲解函数体时提到过主函数 main 的有关内容，下面对 main 函数的参数进行介绍。

在运行程序时，有时需要将必要的参数传递给主函数。主函数 main 的形式参数如下：

```
main(int argc, char* argv[])
```

两个特殊的内部形参 argc 和 argv 是用来接收命令行实参的，这是只有主函数 main 才具有的参数。

☑ argc 参数保存命令行的参数个数，是整型变量。这个参数的值至少是 1，因为至少程序名就是第一个实参。

☑ argv 参数是一个指向字符指针数组的指针，这个数组中的每一个元素都指向命令行实参。所有命令行实参都是字符串，任何数字都必须由程序转变成为适当的格式。

【例 9.10】　main 函数的参数使用。（**实例位置：资源包\TM\sl\9\10**）

在本实例中，通过使用 main 函数的参数，输出程序的位置。

```
#include<stdio.h>

int main(int argc,char* argv[])
{
    printf("%s\n",argv[0]);                /*输出程序的位置*/
    return 0;                              /*程序结束*/
}
```

运行程序，显示效果如图 9.14 所示。



图 9.14　main 函数的参数使用



9.5 函数的调用

在生活中，为了能完成某项特殊的工作，需要使用特定功能的工具。首先要去制作这个工具，工具制作完成后，就要进行使用。函数就像要完成某项功能的工具，而使用函数的过程就是函数的调用。

9.5.1 函数的调用方式

一种工具不只有一种使用方式，函数的调用也是如此。函数的调用方式有 3 种，包括函数语句调用、函数表达式调用和函数参数调用。下面对这 3 种情况进行介绍。

1. 函数语句调用

把函数的调用作为一个语句就称为函数语句调用。函数语句调用是最常用的函数调用方式，如下所示：

```
Display();                                    /*显示一条消息*/
```

这个函数的功能是在函数的内部显示一条消息，这时不要求函数带返回值，只要求完成一定的操作。

【例 9.11】 函数语句调用。(**实例位置：资源包\TM\sl\9\11**)

本实例使用函数语句调用方式，通过调用函数完成显示一条信息的功能，进而观察函数语句调用的使用方式。

```
#include<stdio.h>

void Display()                                /*定义函数*/
{
    printf("Just show this message.");        /*实现显示一条信息的功能*/
}

int main()
{
    Display();                                /*函数语句调用*/
    return 0;                                 /*程序结束*/
}
```

说明

在介绍定义与声明时曾介绍过，如果在使用函数之前定义函数，那么此时的函数定义包含函数声明。

运行程序，显示效果如图 9.15 所示。



图 9.15　函数语句调用

2. 函数表达式调用

函数出现在一个表达式中，这时要求函数必须带回一个确定的值，而这个值则作为参加表达式运算的一部分。如下述代码所示：

```
iResult=iNum3*AddTwoNum(3,5);                /*函数在表达式中，这时 AddTwoNum(3,5)位置应该为具体的值*/
```

可以看到，函数 AddTwoNum 在这条语句中的功能是使两个数相加。在表达式中，AddTwoNum 将相加的结果与 iNum3 变量执行乘法，将得到的结果赋值给 iResult 变量。

【例 9.12】　函数表达式调用。（**实例位置：资源包\TM\sl\9\12**）

在本实例中，定义一个函数，其功能是进行加法计算，并在表达式中调用该函数，使得函数的返回值参加运算，得到新的结果。

```
#include<stdio.h>

/*声明函数，函数进行加法计算*/
int AddTwoNum(int iNum1, int iNum2);

int main()
{
    int iResult;                             /*定义变量，用来存储计算结果*/
    int iNum3=10;                            /*定义变量，赋值为 10*/
    iResult=iNum3*AddTwoNum(3,5);            /*在表达式中调用 AddTwoNum 函数*/
    printf("The result is : %d\n",iResult); /*将计算结果进行输出*/
    return 0;                                /*程序结束*/
}

int AddTwoNum(int iNum1, int iNum2)          /*定义函数*/
{
    int iTempResult;                         /*定义整型变量*/
    iTempResult=iNum1+iNum2;                 /*进行加法计算，并将结果赋值给 iTempResult*/
    return iTempResult;                      /*返回计算结果*/
}
```

（1）在程序代码中，先对要使用的函数进行声明操作。

（2）在主函数 main 中，首先定义整型变量用来保存计算结果。定义整型变量 iNum3，为其赋值 10。

（3）在表达式中调用 AddTwoNum 函数来计算数值 3 和 5 的加法，并且将运算结果赋值给表达式中的元素。iNum3 变量乘以函数返回的值，最后将结果赋值给 iResult 变量。

（4）使用 printf 函数对所得到的结果进行输出显示。

运行程序，显示效果如图 9.16 所示。

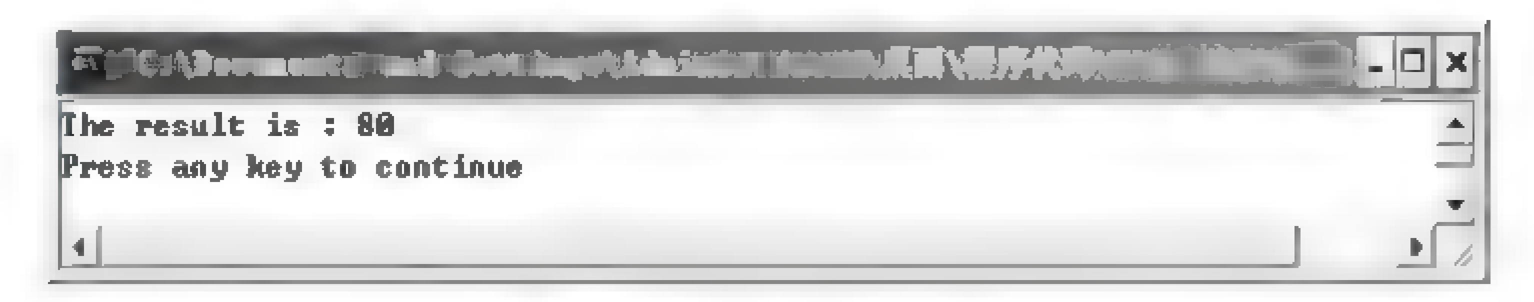


图 9.16　函数表达式调用

3．函数参数调用

函数调用作为一个函数的实参，也就是将函数返回值作为实参，传递到函数中使用。

函数出现在一个表达式中，这时要求函数带回一个确定的值，这个值用于参加表达式的运算。如下代码所示：

```
iResult=AddTwoNum(10,AddTwoNum(3,5));          /*函数在参数中*/
```

在这条语句中，AddTwoNum 函数的功能还是进行两个数相加，然后将相加的结果作为函数的参数，继续进行计算。

【例 9.13】　函数参数调用。（**实例位置：资源包\TM\sl\9\13**）

本实例在前面程序的基础上进行修改，进行连续加法的操作。

```
#include<stdio.h>

/*声明函数，函数进行加法计算*/
int AddTwoNum(int iNum1, int iNum2);

int main()
{
    int iResult;                                  /*定义变量用来存储计算结果*/

    iResult=AddTwoNum(10,AddTwoNum(3,5));         /*在参数中调用 AddTwoNum 函数*/
    printf("The result is : %d\n",iResult);       /*将计算结果进行输出*/
    return 0;                                     /*程序结束*/
}

int AddTwoNum(int iNum1, int iNum2)               /*定义函数*/
{
    int iTempResult;                              /*定义整型变量*/
    iTempResult=iNum1+iNum2;                      /*进行加法计算，并将结果赋值给 iTempResult*/
    return iTempResult;                           /*返回计算结果*/
}
```

在程序中可以看到 AddTwoNum 函数作为函数的参数进行加法操作。

运行程序，显示效果如图 9.17 所示。

```
The result is : 18
Press any key to continue
```

图 9.17　函数参数调用

9.5.2　嵌套调用

在 C 语言中，函数的定义都是互相平行、独立的。也就是说，在定义函数时，一个函数体内不能包含定义的另一个函数，这一点和 Pascal 语言是不同的（Pascal 允许在定义一个函数时，在其函数体内包含另一个函数的定义，而这种形式称为嵌套定义）。例如，下面的代码是错误的：

```
int main()
{
    void Display()                              /*错误！！！不能在函数内定义函数*/
    {
        printf("I want to show the Nesting function");
    }
    return 0;
}
```

从上面的代码中可以看到，在主函数 main 中定义了一个 Display 函数，目的是输出一句提示。但 C 语言是不允许进行嵌套定义的，因此进行编译时就会出现如图 9.18 所示的错误提示。

```
error C2143: syntax error : missing ';' before '{'
```

图 9.18　错误提示

虽然 C 语言不允许进行嵌套定义，但是可以嵌套调用函数，也就是说，在一个函数体内可以调用另外一个函数。例如，使用下面代码进行函数的嵌套调用：

```
void ShowMessage()                              /*定义函数*/
{
    printf("The ShowMessage function");
}

void Display()
{
    ShowMessage();                              /*正确，在函数体内进行函数的嵌套调用*/
}
```

用一个比喻来理解。某公司的 CEO 决定该公司要完成某个目标，但是要完成这个目标就需要将其讲给公司的经理们听，公司的经理们再将要做的内容传递给下级的副经理们听，副经理再讲给下属的职员听，职员按照上级的指示进行工作，最终完成目标，其过程如图 9.19 所示。

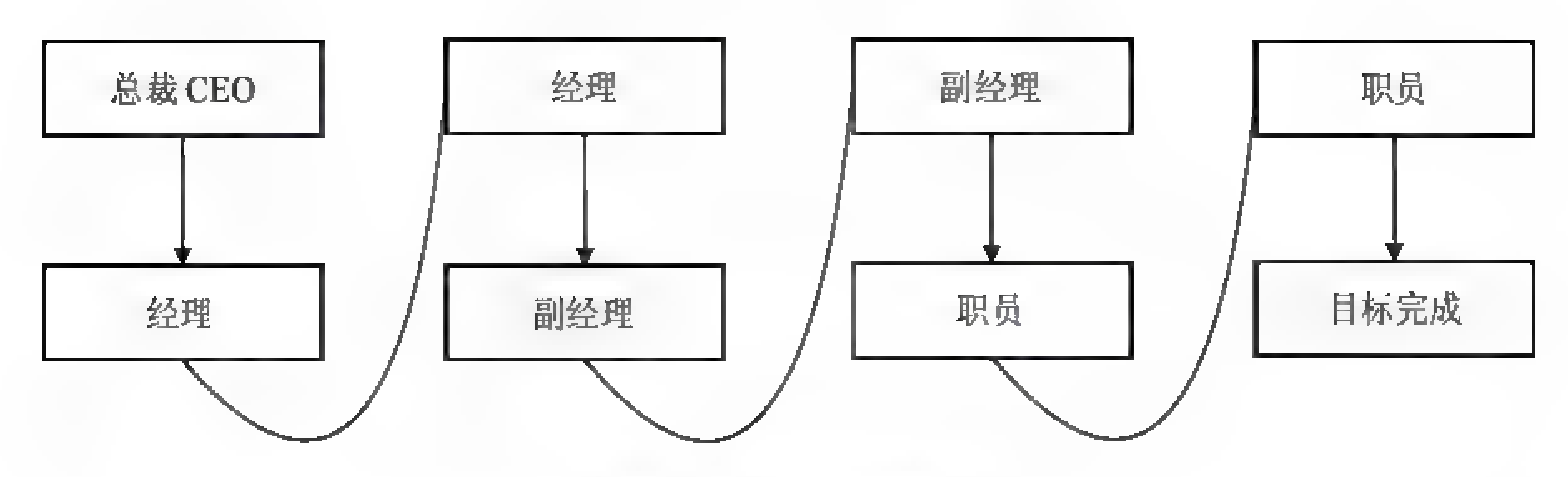


图 9.19　嵌套过程图

【例 9.14】　函数的嵌套调用。（**实例位置：资源包\TM\sl\9\14**）

在本实例中，利用嵌套函数模拟上述比喻中描述的过程，其中将不同层面的人要做的事情封装成不同的函数，通过调用函数完成最终目标。

```
#include<stdio.h>

void CEO();                                    /*声明函数*/
void Manager();
void AssistantManager();
void Clerk();

int main()
{
    CEO();                                     /*调用 CEO 的功能函数*/
    return 0;
}

void CEO()
{
                /*输出信息，表示调用 CEO 函数进行相应的操作*/
    printf("The CEO's working is telling Manager\n");
    Manager();                                 /*调用 Manager 的功能函数*/
}

void Manager()
{
                /*输出信息，表示调用 Manager 函数进行相应的操作*/
    printf("The Manager's working's work is telling AssistantManager\n");
    AssistantManager();                        /*调用 AssistantManager 的功能函数*/
}

void AssistantManager()
{
                /*输出信息，表示调用 AssistantManager 函数进行相应的操作*/
    printf("The AssistantManager's work is telling Clerk\n");
    Clerk();                                   /*调用 Clerk 的功能函数*/
}

void Clerk()
```

```
{
            /*输出信息，表示调用 Clerk 函数进行相应的操作*/
    printf("The Clerk's work is making it\n");
}
```

（1）首先在程序中声明将要使用的函数，其中的 CEO 代表公司总裁，Manager 代表经理，AssistantManager 代表副经理，Clerk 代表职员。

（2）main 函数的下面是有关函数的定义。先来看一下 CEO 函数，通过输出一条信息来表示这个函数的功能和作用，最后在函数体中嵌套调用了 Manager 函数。Manager 和 CEO 函数运行的步骤是相似的，只是最后又在其函数体内调用了 AssistantManager 函数。在 AssistantManager 函数中调用了 Clerk 函数。

（3）在主函数 main 中，调用了 CEO 函数，于是程序的整个流程按照步骤（2）进行，直到 return 0 语句返回，程序结束。

运行程序，显示效果如图 9.20 所示。

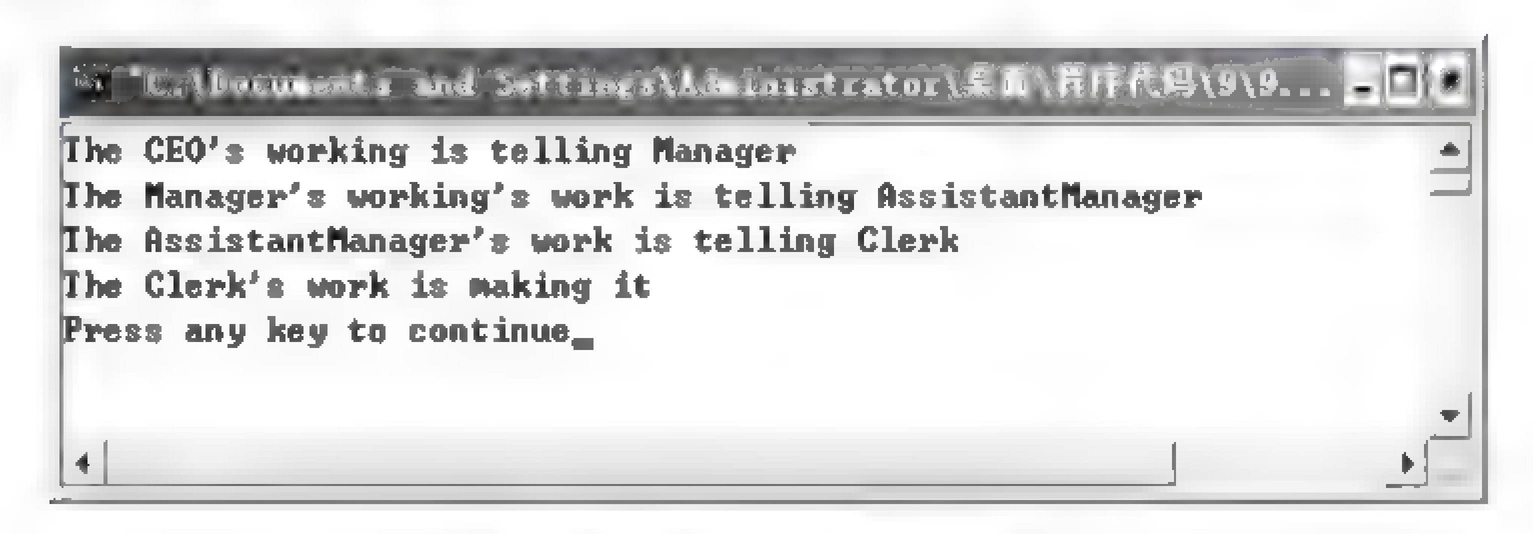


图 9.20　函数的嵌套调用

9.5.3　递归调用

C 语言的函数都支持递归，也就是说，每个函数都可以直接或者间接地调用自己。所谓的间接调用，是指在递归函数调用的下层函数中再调用自己。递归关系如图 9.21 所示。

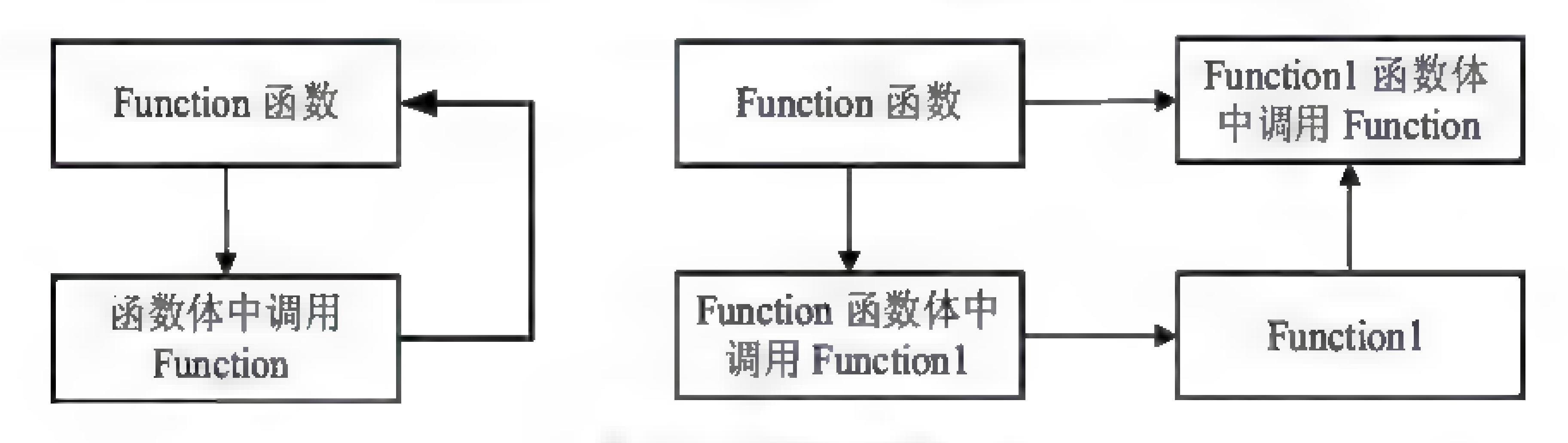


图 9.21　递归调用过程

递归之所以能实现，是因为函数的每个执行过程在栈中都有自己的形参和局部变量的副本，这些副本和该函数的其他执行过程不发生关系。

这种机制是当代大多数程序设计语言实现子程序结构的基础，也使得递归成为可能。假定某个调用函数调用了一个被调用函数，再假定被调用函数又反过来调用了调用函数，那么第二个调用就称为调用函数的递归，因为它发生在调用函数的当前执行过程运行完毕之前。而且，因为原先的调用函数、现在的被调用函数在栈中较低的位置有它独立的一组参数和自变量，原先的参数和变量将不受任何影响，所以递归能正常工作。

【例 9.15】　函数的递归调用。（**实例位置：资源包\TM\sl\9\15**）

在本实例中，定义一个字符串数组，为其赋值一系列的名称，通过递归函数的调用，最后实现逆序显示排列的名单。

```
#include<stdio.h>

void DisplayNames(char** cNameArray);            /*声明函数*/

char* cNames[]=                                  /*定义字符串数组*/
{
    "Aaron",                                     /*为字符串进行赋值*/
    "Jim",
    "Charles",
    "Sam",
    "Ken",
    "end"                                        /*设定结束标志*/
};

int main()
{
    DisplayNames(cNames);                        /*调用递归函数*/
    return 0;
}

void DisplayNames(char** cNameArray)
{
    if(*cNameArray=="end")                       /*判断结束标志*/
    {
        return ;                                 /*函数结束返回*/
    }
    else
    {
        DisplayNames(cNameArray+1);              /*调用递归函数*/
        printf("%s\n",*cNameArray);              /*输出字符串*/
    }
}
```

如图 9.22 所示为程序的流程，通过此图先了解一下程序流程后再进行讲解，会使读者对程序有一个更清晰的认识。

对程序进行分析：

（1）源文件中首先声明要用到的递归函数，递归函数的参数声明为指针的指针。

（2）定义一个全局字符串数组，并且为其进行赋值。其中的一个字符串数组元素“end”作为字符串数组的结尾标志。

（3）在主函数 main 中调用递归函数 DisplayNames。

（4）在源文件的下面是有关 DisplayNames 函数的定义。在 DisplayNames 的函数体中，通过一个 if 语句判断此时要输出的字符串是否是结束字符，如果是结束标志“end”字符，那么使用 return 语句进行返回。如果不满足要求，则执行下面的 else 语句，在语句块中先调用的是递归函数，在函数参数

处可以看到传递的字符串数组元素发生改变，传递下一个数组元素。如果调用递归函数，则又开始判断传递进来的字符串是否是数组的结束标志。最后输出字符串数组的元素。

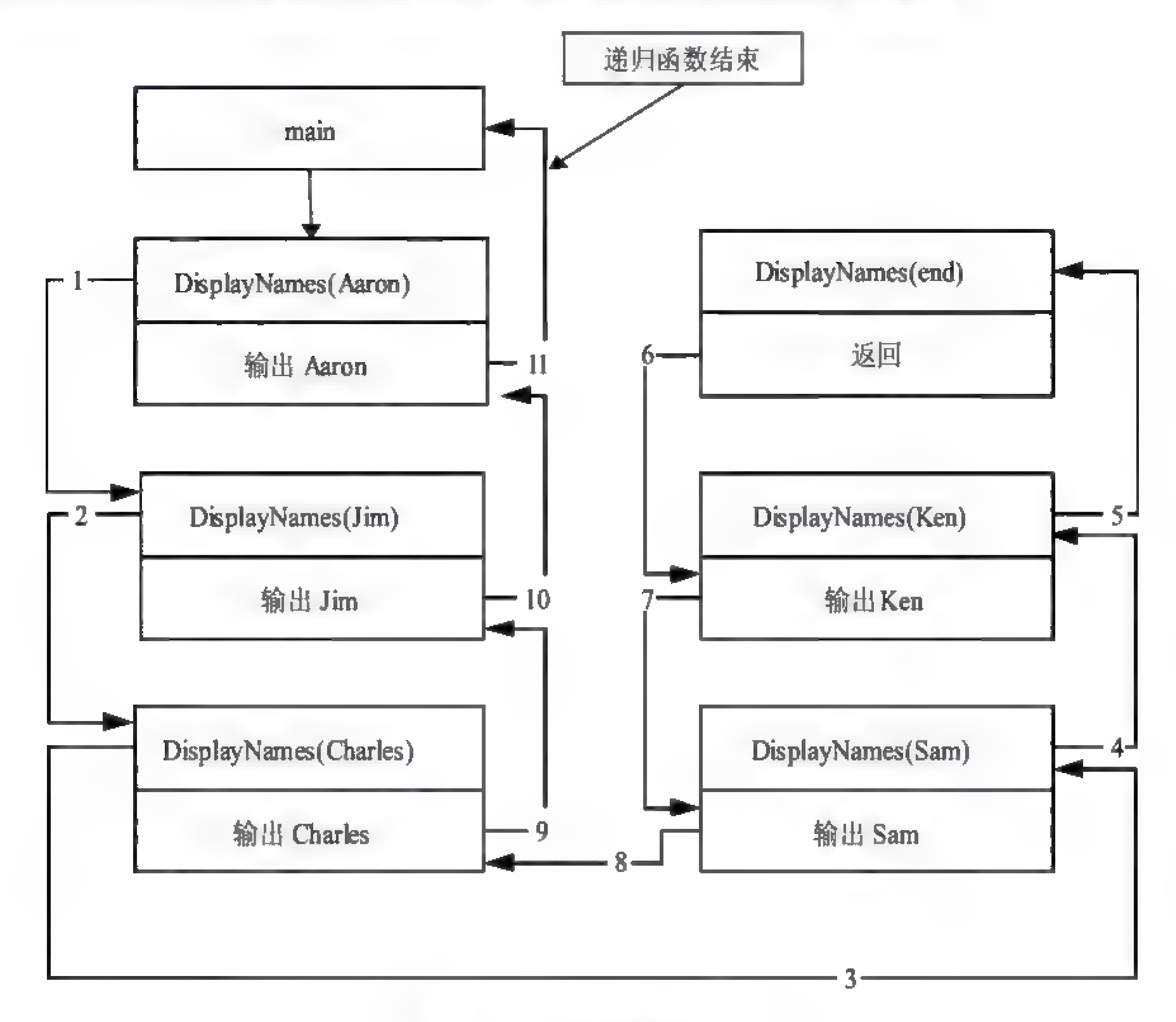


图 9.22　程序调用流程图

运行程序，显示效果如图 9.23 所示。



图 9.23　函数的递归调用



9.6　内部函数和外部函数

函数是 C 语言程序中的最小单位，可以把一个函数或多个函数保存为一个文件，这个文件称为源文件。定义一个函数后，这个函数就可以被另外的函数所调用。但当一个源程序由多个源文件组成时，可以指定函数不能被其他文件调用。这样，C 语言又把函数分为两类：一类是内部函数，另一类是外部函数。

9.6.1 内部函数

定义一个函数，如果希望这个函数只被所在的源文件使用，那么就称这样的函数为内部函数。内部函数又称为静态函数。使用内部函数，可以使函数只局限在函数所在的源文件中，如果在不同的源文件中有同名的内部函数，则这些同名的函数是互不干扰的。

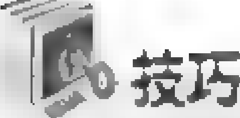
在定义内部函数时，要在函数返回值和函数名前面加上关键字 static 进行修饰：

```
static  返回值类型  函数名(参数列表)
```

例如，定义一个功能是进行加法运算且返回值是 int 型的内部函数，代码如下：

```
static int Add(int iNum1,int iNum2)
```

在函数的返回值类型 int 前加上关键字 static，可将原来的函数修饰成内部函数。



技巧

使用内部函数的好处是，不同的开发者可以分别编写不同的函数，而不必担心所使用的函数是否会与其他源文件中的函数同名，因为内部函数只可以在所在的源文件中进行使用，所以即使不同的源文件中有相同的函数名，也没有关系。

【例 9.16】 内部函数的使用。（**实例位置：资源包\TM\sl\9\16**）

在本实例中使用内部函数，通过一个函数对字符串进行赋值，再通过一个函数对字符串进行输出显示。

```
#include<stdio.h>

static char* GetString(char* pString)            /*定义赋值函数*/
{
    return pString;                              /*返回字符*/
}

static void ShowString(char* pString)            /*定义输出函数*/
{
    printf("%s\n",pString);                      /*显示字符串*/
}

int main()
{
    char* pMyString;                             /*定义字符串变量*/

    pMyString=GetString("Hello!");               /*调用函数为字符串赋值*/
    ShowString(pMyString);                       /*显示字符串*/

    return 0;
}
```

在程序中，使用 static 关键字对函数进行修饰，使其只能在其源文件中进行调用。

运行程序，显示效果如图 9.24 所示。



图 9.24　内部函数的使用

9.6.2　外部函数

与内部函数相反的就是外部函数，外部函数是可以被其他源文件调用的函数。定义外部函数使用关键字 extern 进行修饰。在使用一个外部函数时，要先用 extern 声明所用的函数是外部函数。

例如，函数头可以写成下面的形式：

```
extern int Add(int iNum1,int iNum2);
```

这样，Add 函数就可以被其他源文件调用，进行加法运算。

注意

在 C 语言中定义函数时，如果不指明函数是内部函数还是外部函数，那么将默认指定函数为外部函数，也就是说，定义外部函数时可以省略关键字 extern。书中的多数实例所使用的函数都为外部函数。

【例 9.17】　外部函数的使用。（**实例位置：资源包\TM\sl\9\17**）

在本实例中，使用外部函数完成和例 9.16 中使用内部函数时相同的功能，只是所用的函数不包含在同一个源文件中。

```
/*//////////////////////////////////////////////////////////////////////////////////////////////*/
/*                              ExternFun.c                                    */
/*//////////////////////////////////////////////////////////////////////////////////////////////*/
#include<stdio.h>

extern char* GetString(char* pString);          /*声明外部函数*/
extern void ShowString(char* pString);          /*声明外部函数*/

int main()
{
    char* pMyString;                            /*定义字符串变量*/
    pMyString=GetString("Hello!");              /*调用函数为字符串赋值*/
    ShowString(pMyString);                      /*显示字符串*/

    return 0;
}
```

```
/*//////////////////////////////////////////////////////////////////////////////////////////////*/
/*                              ExternFun1.c                                   */
/*//////////////////////////////////////////////////////////////////////////////////////////////*/
```

```
extern char* GetString(char* pString)
{
    return pString;                          /*返回字符*/
}
```

```
/*////////////////////////////////////////////////////////////////////////////////////////////////*/
/*                          ExternFun2.c                                        */
/*////////////////////////////////////////////////////////////////////////////////////////////////*/
extern void ShowString(char* pString)
{
    printf("%s\n",pString);                  /*显示字符串*/
}
```

从上面的程序中，可以看到代码和例 9.16 几乎是相同的，但是由于使用 extern 关键字使得函数为外部函数，因此可以将函数放入其他源文件中。

（1）主函数 main 在源文件 ExternFun.c 中。首先声明两个函数，其中使用 extern 关键字说明函数为外部函数。然后在 main 函数体中调用这两个函数，GetString 函数对 pMyString 变量进行赋值，而 ShowString 函数用来输出变量。

（2）在 ExternFun1.c 源文件中对 GetString 函数进行定义，通过对传入的参数执行返回操作，完成对变量的赋值功能。

（3）在 ExternFun2.c 源文件中对 ShowString 函数进行定义，在函数体中使用 printf 函数对传递进来的参数进行显示。

运行程序，显示效果如图 9.25 所示。



图 9.25　外部函数的使用

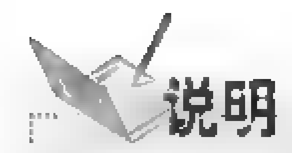


9.7　局部变量和全局变量

在讲解有关局部变量和全局变量的知识之前，先来了解一些有关作用域方面的知识。作用域决定了程序中的哪些语句是可用的，换句话说，就是在程序中的可见性。作用域包括局部作用域和全局作用域，局部变量具有局部作用域，全局变量具有全局作用域。下面具体看一下有关局部变量和全局变量的内容。

9.7.1　局部变量

在一个函数的内部定义的变量是局部变量。上述实例中绝大多数的变量都只是局部变量，这些变量声明在函数内部，无法被其他函数所使用。函数的形式参数也属于局部变量，作用范围仅限于函数内部的所有语句块。



说明

在语句块内声明的变量仅在该语句块内部起作用，当然也包括嵌套在其中的子语句块。

图 9.26 表示的是不同情况下局部变量的作用域范围。

```
int Function1(int iA)
{
        ...
}
```
iA 的作用域范围

```
float Function2(int iB)
{
        float fB1,fB2;
        ...
}
```
iB、fB1 和 fB2 的作用域范围

```
int main()
{
        int iC;
        float fC1,fC2;
        ...
        return 0;
}
```
iC、fC1 和 fC2 的作用域范围

```
int main()
{
        int iD;
        for(iD=1;iD<10;iD++)
        {
                char cD;
                ...
        }
        return 0;
}
```
cD 的作用域范围

iD 的作用域范围

图 9.26　局部变量的作用域范围

【例 9.18】　局部变量的作用域。（**实例位置：资源包\TM\sl\9\18**）

本实例在不同的位置定义一些变量，并为其赋值来表示变量的所在位置，最后输出显示其变量值，通过输出的信息来观察局部变量的作用范围。

```
#include<stdio.h>

int main()
{
    int iNumber1=1;                          /*iNumber1 的作用域在整个 main 函数中*/
    if(iNumber1>0)
    {
        int iNumber2=2;                      /*iNumber2 的作用域在 if 语句块中*/
        if(iNumber2>0)
        {
```

```
            int iNumber3=3;                       /*iNumber3 的作用域在 if 语句块中*/
                                                  /*将 3 个都在此作用域的函数进行输出*/
            printf("All three number are in scope here %d   %d   %d\n",
                iNumber1,iNumber2,iNumber3);
        }
    }
    return 0;
}
```

程序中有3个作用域范围，主函数main是其中最大的作用域范围，因为定义变量iNumber1在main函数中，所以iNumber1的范围是在整个main函数体中。而iNumber2定义在第一个if语句块中，因此它的使用范围就是在第一个if语句块内。变量iNumber3在最内部的嵌套层，因此使用范围只在最里面的if语句块中。

从上面的描述中可以看到，一个局部变量的作用范围可以由包含变量的一对大括号所限定，这样就可以更好地观察局部变量的作用域。

运行程序，显示效果如图9.27所示。



图9.27　局部变量的作用域

在C语言中，位于不同作用域的变量可以使用相同的标识符，也就是可以为变量起相同的名称。此时读者朋友们有没有想到这样一种情况：如果内层作用域中定义的变量和已经声明的某个外层作用域中的变量有相同的名称，在内层中使用这个变量名，那么此时这个变量名表示的是外层变量还是内层变量呢？答案是：内层作用域中的变量将屏蔽外层作用域中的那个变量，直到结束内层作用域为止。这就是局部变量的屏蔽作用。

【例9.19】　局部变量的屏蔽作用。（**实例位置：资源包\TM\sl\9\19**）

在本实例中，不同的语句块定义了3个相同名称的变量，通过输出变量值来演示有关局部变量的屏蔽作用效果。

```
#include<stdio.h>

int main()                                        /*主函数 main*/
{
    int iNumber1=1;                               /*在第一个 iNumber1 定义变量*/
    printf("%d\n",iNumber1);                      /*输出变量值*/

    if(iNumber1>0)
    {
        int iNumber1=2;                           /*在第二个 iNumber1 定义变量*/
        printf("%d\n",iNumber1);                  /*输出变量值*/

        if(iNumber1>0)
```

```
        {
            int iNumber1=3;                    /*在第 3 个 iNumber1 定义变量*/
            printf("%d\n",iNumber1);           /*输出变量值*/
        }

        printf("%d\n",iNumber1);               /*输出变量值*/
    }

    printf("%d\n",iNumber1);                   /*输出变量值*/
    return 0;
}
```

通过运行程序对得到的显示结果进行分析：

（1）在主函数 main 中，定义了第一个整型变量 iNumber1，将其赋值为 1，赋值之后使用 printf 函数进行输出变量 iNumber1。在程序的运行结果中可以看到，此时 iNumber1 的值为 1。

（2）使用 if 语句进行判断，这里使用 if 语句的目的在于划分出一段语句块。因为位于不同作用域的变量可以使用相同的标识符，所以在 if 语句块中也定义一个 iNumber1 变量，并将其赋值为 2。再次使用 printf 函数输出变量 iNumber1 的操作，观察一下程序的运行结果，发现第二个输出的值为 2。此时值为 2 的变量在此作用域中就将值为 1 的变量屏蔽掉。

（3）在 if 语句中再次进行嵌套，其嵌套语句中定义相同标识符的 iNumber1 变量，为了进行区分，将其赋值为 3。调用 printf 函数输出变量 iNumber1，从程序运行的结果可以看出显示结果为 3。由此看出值为 3 的变量将值为 2 与 1 的两个变量都进行了屏蔽。

（4）在最深层嵌套的 if 语句结束之后，使用 printf 函数进行输出，发现此时显示的值为 2。由此说明此时已经不在值为 3 的变量作用域范围，而在值为 2 的作用域范围。

（5）当 if 语句结束之后，输出变量值，此时显示的变量值为 1，说明离开了值为 2 的作用域范围，不再对值为 1 的变量产生变量的屏蔽作用。

运行程序，显示效果如图 9.28 所示。

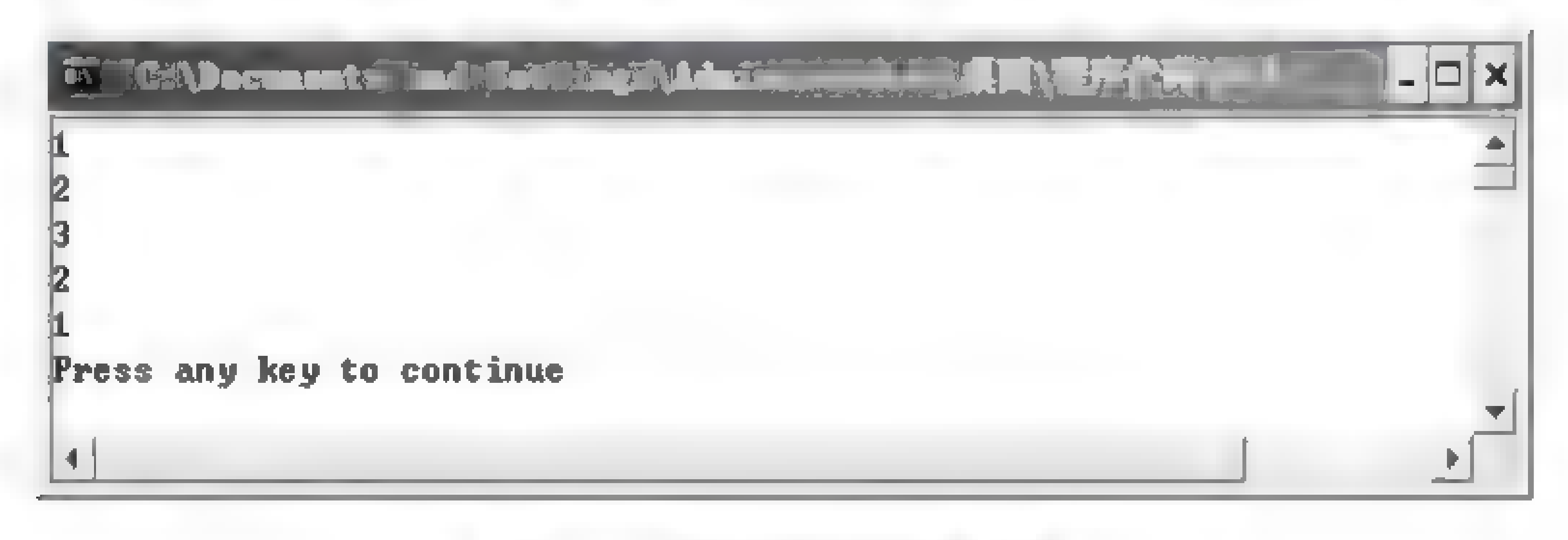


图 9.28　局部变量的屏蔽作用

9.7.2　全局变量

程序的编译单位是源文件，通过上文的介绍可以了解到在函数中定义的变量称为局部变量。如果一个变量在所有函数的外部声明，这个变量就是全局变量。顾名思义，全局变量是可以在程序的任何位置进行访问的变量。

注意

全局变量不属于某个函数，而属于整个源文件。如果外部文件要进行使用，则要用 extern 关键字进行引用修饰。

定义全局变量的作用是增加函数间数据联系的渠道。由于同一个文件中的所有函数都能引用全局变量的值，因此如果在一个函数中改变了全局变量的值，就能影响到其他函数，相当于各个函数间有直接传递通道。

例如，有一家全国连锁商店机构，商店所使用的价格是全国统一的。全国各地有很多这样的连锁商店，当进行价格调整时，应该确保每一家连锁商店的价格是相同的。全局变量就像其中所要设定的价格，而函数就像每一家连锁店，当全局变量进行修改时，那么函数中使用的该变量都将被更改。

为了使读者更为清楚地掌握其概念，使用下面的实例模拟上面的比喻。

【例 9.20】　使用全局变量模拟价格调整。（**实例位置：资源包\TM\sl\9\20**）

在本程序中，使用全局变量模拟连锁店的全国价格调整，使用函数表示连锁店，并在函数中输出一条消息，表示连锁店中的价格。

```
#include<stdio.h>

int iGlobalPrice=100;                    /*设定商店的初始价格*/

void Store1Price();                      /*声明函数，代表第 1 个连锁店*/
void Store2Price();                      /*代表第 2 个连锁店*/
void Store3Price();                      /*代表第 3 个连锁店*/
void ChangePrice();                      /*更改连锁店的统一价格*/

int main()
{
        /*先显示价格改变之前所有连锁店的价格*/
    printf("the chain store's original price is : %d\n",iGlobalPrice);
    Store1Price();                       /*显示 1 号连锁店的价格*/
    Store2Price();                       /*显示 2 号连锁店的价格*/
    Store3Price();                       /*显示 3 号连锁店的价格*/
        /*调用函数，改变连锁店的价格*/
    ChangePrice();
        /*显示提示，显示修改后的价格*/
    printf("the chain store's   present price is : %d\n",iGlobalPrice);
    Store1Price();                       /*显示 1 号连锁店的当前价格*/
    Store2Price();                       /*显示 2 号连锁店的当前价格*/
    Store3Price();                       /*显示 3 号连锁店的当前价格*/
    return 0;
}
/*定义 1 号连锁店的价格函数*/
void Store1Price()
{
    printf("store1's price is : %d\n",iGlobalPrice);
}
```

```
/*定义 2 号连锁店的价格函数*/
void Store2Price()
{
	printf("store2's price is : %d\n",iGlobalPrice);
}
/*定义 3 号连锁店的价格函数*/
void Store3Price()
{
	printf("store3's price is : %d\n",iGlobalPrice);
}
/*定义更改连锁店价格函数*/
void ChangePrice()
{
	printf("What price do you want to change?   the price is: ");
	scanf("%d",&iGlobalPrice);
}
```

（1）在程序中，定义了一个全局变量 iGlobalPrice 来表示所有连锁店的价格，为了可以形成对比，初始化值为 100。定义的一种函数代表连锁店的价格，例如 Store1Price 代表 1 号连锁店；定义的另一种函数用来改变全局变量的值，也就代表了对所有连锁店进行调价。

（2）在主函数 main 中，首先是将连锁店的先前价格进行显示，之后通过一条信息提示更改 iGlobal 变量。当全局变量被修改后，将所有连锁店当前的价格再次进行输出和对比。

（3）通过程序的运行结果可以看出，全局变量增加了函数间数据联系的渠道，当修改一个全局变量时，所有函数中的该变量都会发生改变。

运行程序，显示效果如图 9.29 所示。

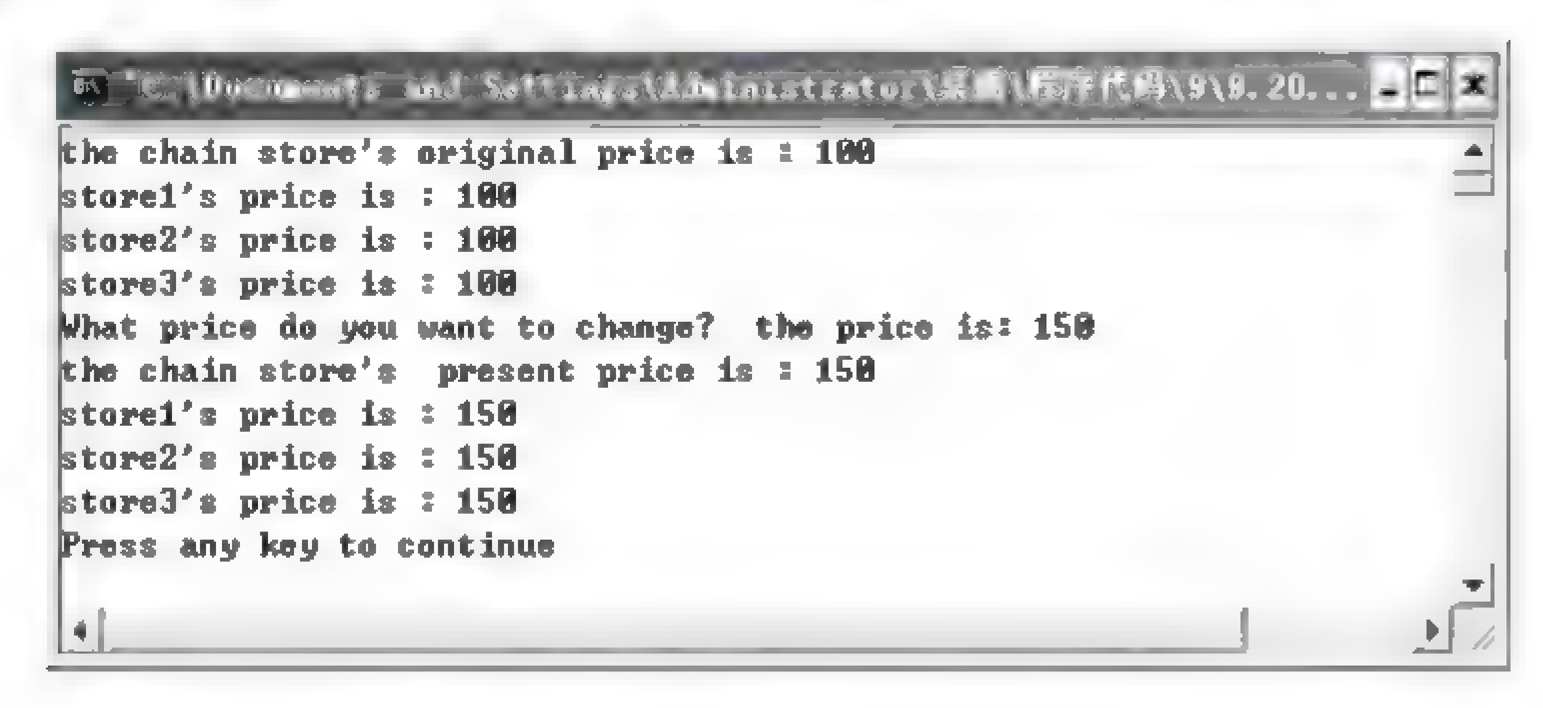


图 9.29　使用全局变量模拟价格调整



9.8　函 数 应 用

为了使用户能快速编写程序，编译系统通常会提供一些库函数，以供用户调用。不同的编译系统，其所提供的库函数可能不完全相同，可能函数名称相同，但是实现的功能不同；也有可能实现统一功能，但是函数的名称不同。ANSI C 标准建议提供的标准库函数包括了目前多数 C 编译系统所提供的库函数，下面就介绍一部分常用的库函数。

在程序中经常会使用一些数学的运算或者公式，这里首先介绍有关数学的常用函数。

1．abs 函数

该函数的功能是求整数的绝对值。函数定义如下：

```
int abs(int i);
```

例如，求一个负数的绝对值的方法如下：

```
int iAbsoluteNumber;                    /*定义整数*/
int iNumber = -12;                      /*定义整数，为其赋值-12*/
iAbsoluteNumber=abs(iNumber);           /*将 iNumber 的绝对值赋给 iAbsoluteNumber 变量*/
```



注意

在使用数学函数时，要为程序添加头文件#include<math.h>。

2．labs 函数

该函数的功能是求长整数的绝对值。函数定义如下：

```
long labs(long n);
```

例如，求一个长整型的绝对值的方法如下：

```
long lResult;                           /*定义长整型*/
long lNumber = -1234567890L;            /*定义长整型，为其赋值-1234567890*/
lResult= labs(lNumber);                 /*将 lNumber 的绝对值赋给 lResult 变量*/
```

3．fabs 函数

该函数的功能是返回浮点数的绝对值。函数定义如下：

```
double fabs(double x);
```

例如，求一个实型的绝对值的方法如下：

```
double fFloatResult;                    /*定义实型变量*/
double fNumber = -1234.0;               /*定义实型变量，为其赋值-1234.0*/
fFloatResult= fabs(fNumber);            /*将 fNumber 的绝对值赋给 fResult 变量*/
```

【例 9.21】　数学库函数使用。（**实例位置：资源包\TM\sl\9\21**）

在本实例中，将上述介绍的 3 个库函数放在一起，通过调用函数，观察函数的作用。

```
#include<stdio.h>
#include<math.h>                        /*包含头文件 math.h*/
int main()
{
    int iAbsoluteNumber;                /*定义整数*/
    int iNumber = -12;                  /*定义整数，为其赋值-12*/
    long lResult;                       /*定义长整型*/
```

```
    long lNumber = -1234567890L;                    /*定义长整型，为其赋值-1234567890*/
    double fFloatResult;                            /*定义浮点型*/
    double fNumber = -123.1;                        /*定义浮点型，为其赋值-1234.0*/

    iAbsoluteNumber=abs(iNumber);                   /*将 iNumber 的绝对值赋给 iAbsoluteNumber 变量*/
    lResult= labs(lNumber);                         /*将 lNumber 的绝对值赋给 lResult 变量*/
    fFloatResult= fabs(fNumber);                    /*将 fNumber 的绝对值赋给 fFloatResult 变量*/

    /*输出原来的数字，然后将得到的绝对值进行输出*/
    printf("the original number is: %d, the absolute is: %d\n",iNumber,iAbsoluteNumber);
    printf("the original number is: %ld, the absolute is: %ld\n",lNumber,lResult);
    printf("the original number is: %lf, the absolute is: %lf\n",fNumber,fFloatResult);

    return 0;
}
```

上述程序代码通过使用数学函数，求取已经赋值完成的变量，并将得到的数值存储在其他变量中，最后使用输出函数将原来的数值和求取后的数值都进行输出。

运行程序，显示效果如图 9.30 所示。

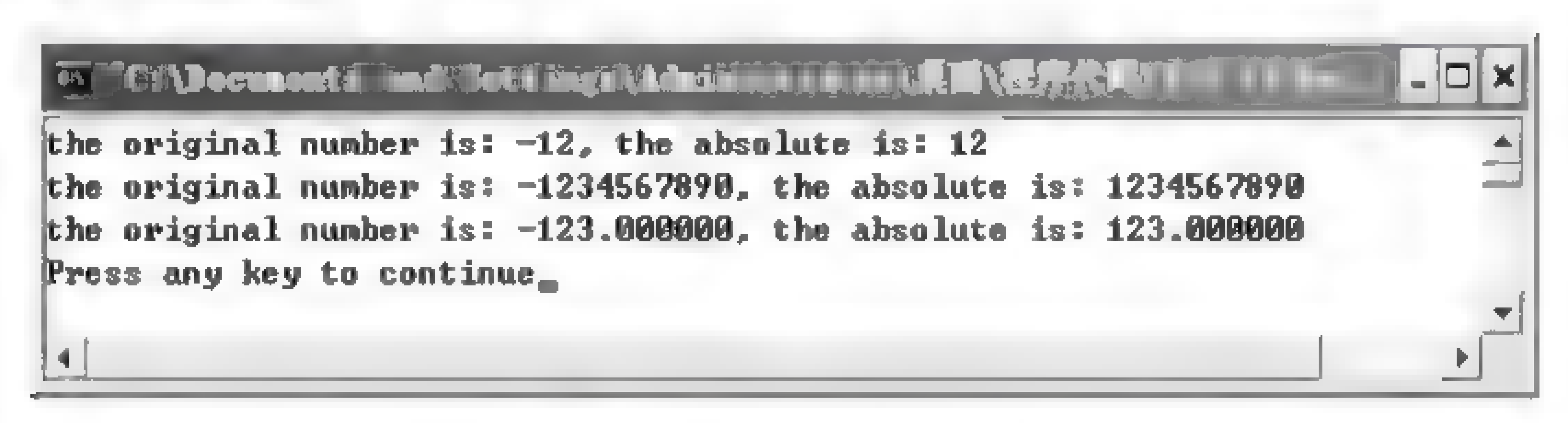


图 9.30　数学库函数使用

4. sin 函数

该函数的功能是求解正弦。函数定义如下：

```
double sin(double x);
```

例如，求正弦值的方法如下：

```
double fResultSin;                                  /*定义实型变量*/
double fXsin = 0.5;                                 /*定义实型变量，并进行赋值*/
fResultSin = sin(fXsin);                            /*使用正弦函数*/
```

5. cos 函数

该函数的功能是求解余弦。函数定义如下：

```
double cos(double x);
```

例如，求余弦值的方法如下：

```
double fResultCos;                                  /*定义实型变量*/
double fXcos = 0.5;                                 /*定义实型变量，为其赋值为 0.5*/
fResultCos = cos(fXcos);                            /*调用余弦函数*/
```

6. tan 函数

该函数的功能是求解正切。函数定义如下：

```
double tan(double x);
```

例如，求正切值的方法如下：

```
double fResultTan;                          /*定义实型变量*/
double fXtan = 0.5;                         /*定义实型变量，为其赋值为 0.5*/
fResultTan = tan(fXtan);                    /*调用正切函数*/
```

【例 9.22】　使用三角函数。（**实例位置：资源包\TM\sl\9\22**）

在本程序中，利用库函数中的数学函数解决有关三角运算的问题。

```
#include<stdio.h>
#include<math.h>                            /*包含头文件 math.h*/

int main()
{
    double fResultSin;                      /*用来保存正弦值*/
    double fResultCos;                      /*用来保存余弦值*/
    double fResultTan;                      /*用来保存正切值*/

    double fXsin =0.5;
    double fXcos = 0.5;
    double fXtan = 0.5;

    fResultSin = sin(fXsin);                /*调用正弦函数*/
    fResultCos = cos(fXcos);                /*调用余弦函数*/
    fResultTan = tan(fXtan);                /*调用正切函数*/
    /*输出运算结果*/
    printf("The sin of %lf is %lf\n", fXsin, fResultSin);
    printf("The cos of %lf is %lf\n", fXcos, fResultCos);
    printf("The tan of %lf is %lf\n", fXtan, fResultTan);
    return 0;
}
```

在使用数学函数时，要先包含头文件 math.h。代码中，先定义用来保存计算结果的变量，之后定义要计算的变量，为了能看出结果的不同，在此都将其赋值为 0.5，然后通过三角函数得到结果，最后通过输出语句将原值和结果都进行输出显示。

运行程序，显示效果如图 9.31 所示。

```
"C:\Documents and Settings\Administrator\桌面\程序代码\9\9.22\Deb...
The sin of 0.500000 is 0.479426
The cos of 0.500000 is 0.877583
The tan of 0.500000 is 0.546302
Press any key to continue_
```

图 9.31　使用三角函数

下面要介绍的是另一类常用的函数，即有关字符和字符串的函数。

7．isalpha 函数

该函数的功能是检测字母，如果参数（ch）是字母表中的字母（大写或小写），则返回非零。使用时要包含头文件 ctype.h。函数定义如下：

```
int isalpha(int ch);
```

例如，判断输入的字符是否为字母的方法如下：

```
char c;                                         /*定义字符变量*/
scanf("%c", &c);                                /*输入字符*/
isalpha(c);                                     /*调用 isalpha 函数判断输入的字符*/
```

8．isdigit 函数

该函数的功能是检测数字，如果 ch 是数字，则函数返回非零值，否则返回零。使用时要包含头文件 ctype.h。函数定义如下：

```
int isdigit(int ch);
```

例如，判断输入的字符是否为数字的方法如下：

```
char c;                                         /*定义字符变量*/
scanf("%c", &c);                                /*输入字符*/
isdigit(c);                                     /*调用 isdigit 函数判断输入的字符*/
```

9．isalnum 函数

该函数的功能是检测字母或数字，如果参数是字母表中的一个字母或是一个数字，则函数返回非零值，否则返回零。使用时要包含头文件 ctype.h。函数定义如下：

```
int isalnum(int ch);
```

例如，判断输入的字符是否为数字或字母的方法如下：

```
char c;                                         /*定义字符变量*/
scanf("%c", &c);                                /*输入字符*/
isalnum(c);                                     /*调用 isalnum 函数判断输入的字符*/
```

【例 9.23】 使用字符函数判断输入字符。（**实例位置：资源包\TM\sl\9\23**）

在本程序中，通过向控制台输入字符，利用 if 判断语句和字符函数判断输入的是哪一种类型的字符，然后根据不同的字符类型输出相应提示信息。

```
#include<stdio.h>
#include<ctype.h>

void SwitchShow(char c);

int main()
{
```

```
    char cCharPut;                          /*定义字符变量，用来接收输入的字符*/
    char cCharTemp;                         /*定义字符变量，用来接收回车符*/


    printf("First enter:");                 /*消息提示，第一次输入字符*/
    scanf( "%c", &cCharPut);                /*输入字符*/
    SwitchShow(cCharPut);                   /*调用函数进行判断*/
    cCharTemp=getchar();                    /*接收回车符*/

    printf("Second enter:");                /*消息提示，第二次输入字符*/
    scanf( "%c", &cCharPut);                /*输入字符*/
    SwitchShow(cCharPut);                   /*调用函数判断输入的字符*/
    cCharTemp=getchar();                    /*接收回车符*/

    printf("Third enter:");                 /*消息提示，第 3 次输入字符*/
    scanf( "%c", &cCharPut);                /*输入字符*/
    SwitchShow(cCharPut);                   /*调用函数判断输入的字符*/

    return 0;                               /*程序结束*/
}

void SwitchShow(char cChar)
{
    if(isalpha(cChar))                      /*判断是否为字母*/
    {
        printf("You entered a letter of the alphabet %c\n",cChar);
    }

    if(isdigit(cChar))                      /*判断是否为数字*/
    {
        printf("You entered the digit %c\n", cChar);
    }

    if(isalnum(cChar))                      /*判断是否为字母或者数字*/
    {
        printf("You entered the alphanumeric character %c\n", cChar);
    }

    else                                    /*当字符既不是字母也不是数字时*/
    {
        printf("You entered the character is not alphabet or digit :%c\n", cChar);
    }
}
```

（1）要使用字符函数，先要引入头文件 ctype.h。

（2）在程序中定义了两个字符变量，cCharPut 用来在程序中接收用户输入的字符，而 cCharTemp 的作用是接收用户按 Enter 键所产生的回车符。

（3）定义 SwitchShow 函数实现在程序中判断字符的功能，这样可以使程序更简洁。在 SwitchShow 函数体中，通过在 if 语句的判断条件中调用字符函数，根据调用字符函数的返回值结果判断传递的字符参数 cChar 是哪一种情况，最后通过在不同情况中的提示信息来表示判断的结果。

（4）在 main 函数中调用了 getchar 函数，其作用是获取一个字符。用户在输入字符时，每次输入完毕后都要按 Enter 键进行确定，这样回车符就会变成下一次要输入的字符，因此，这里调用 getchar 函数将回车符进行提取。

说明

读者可以尝试将 getchar 函数所在行的代码注销掉，运行程序观察结果，会发现其中第二个输入被程序跳过。

运行程序，显示效果如图 9.32 所示。

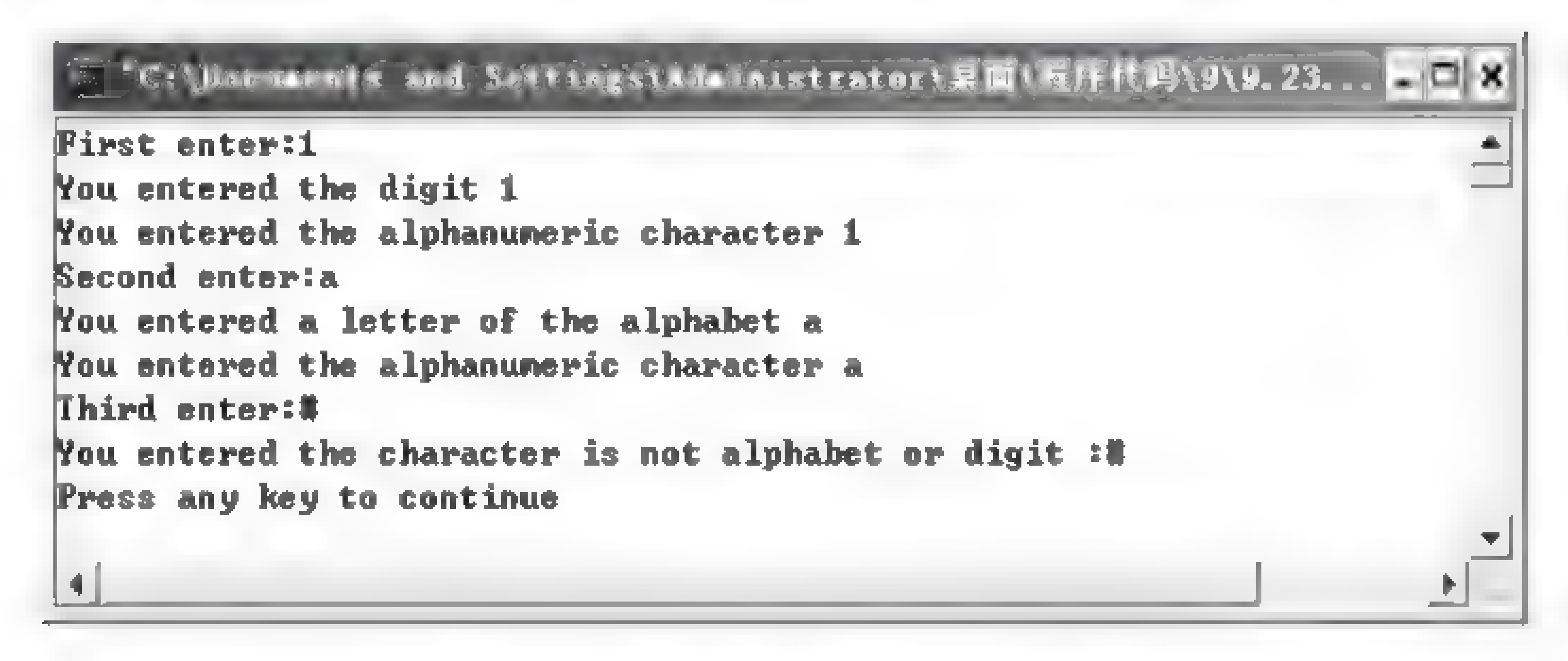


图 9.32　使用字符函数判断输入字符

9.9　小　　结

本章主要讲解 C 语言中函数的相关内容，包括以下方面：

☑　函数的定义。
☑　函数的返回语句。
☑　函数的参数。
☑　函数的调用。
☑　内部函数和外部函数。
☑　局部变量和外部变量。
☑　函数的应用。

首先讲解了函数定义，使读者能够定义一个函数。接着通过对返回语句和函数参数的介绍，使读者更深一步了解函数的细节部分。只知道如何定义函数是不够的，通过介绍函数的调用，将函数的各种调用方式与方法进行了详细的说明，再利用实例使读者有“不仅看得见，并且摸得着”的感觉。接下来讲解内部函数和外部函数，以及局部变量和全局变量的知识，更深入地探讨细节部分。最后讲解一些常用的函数，并通过一些实例进行演示，使读者能轻松地领悟函数的功能。

函数是C语言的重点部分，希望读者对此部分的知识能多加理解。

9.10 实践与练习

1．定义一个标识符为 Max 的函数，其函数功能是判断两个整数的大小，并将较大的整数显示出来。**（答案位置：资源包\TM\sl\9\24）**

2．定义一个一维数组 Score，存放10个元素，代表10个学生的成绩。要求设计函数，将数组名作为函数的参数，函数功能是求出这10个学生的平均成绩。**（答案位置：资源包\TM\sl\9\25）**

第10章

指针

（视频讲解：1小时2分钟）

指针是C语言的一个重要组成部分，是C语言的核心、精髓所在。用好指针，可以在C语言编程中起到事半功倍的效果：一方面，可以提高程序的编译效率、执行速度，以及实现动态的存储分配；另一方面，可使程序更灵活，便于表示各种数据结构，编写高质量的程序。

通过阅读本章，您可以：

- 掌握指针的相关概念
- 掌握数组与指针之间的关系
- 掌握指向指针的指针
- 掌握如何使用指针变量作为函数参数
- 了解main函数的参数



10.1 指针相关概念

指针是C语言显著的优点之一，其使用起来十分灵活而且能提高某些程序的效率，但是如果使用不当则很容易造成系统错误。许多程序"挂死"往往都是错误地使用指针造成的。

10.1.1 地址与指针

系统的内存就好比是带有编号的小房间，如果想使用内存就需要得到房间编号。图10.1定义了一个整型变量i，整型变量需要4个字节，所以编译器为变量i分配的编号为1000～1003。

什么是地址？地址就是内存区中对每个字节的编号，如图10.1所示的1000、1001、1002和1003就是地址。为了进一步说明，来看图10.2。

如图10.2所示的1000、1004等就是内存单元的地址，而0、1就是内存单元的内容，换种说法就是基本整型变量i在内存中的地址从1000开始。因为基本整型占4个字节，所以变量j在内存中的起始地址为1004，变量i的内容是0。

那么指针又是什么呢？这里仅将指针看作是内存中的一个地址，多数情况下，这个地址是内存中另一个变量的位置，如图10.3所示。

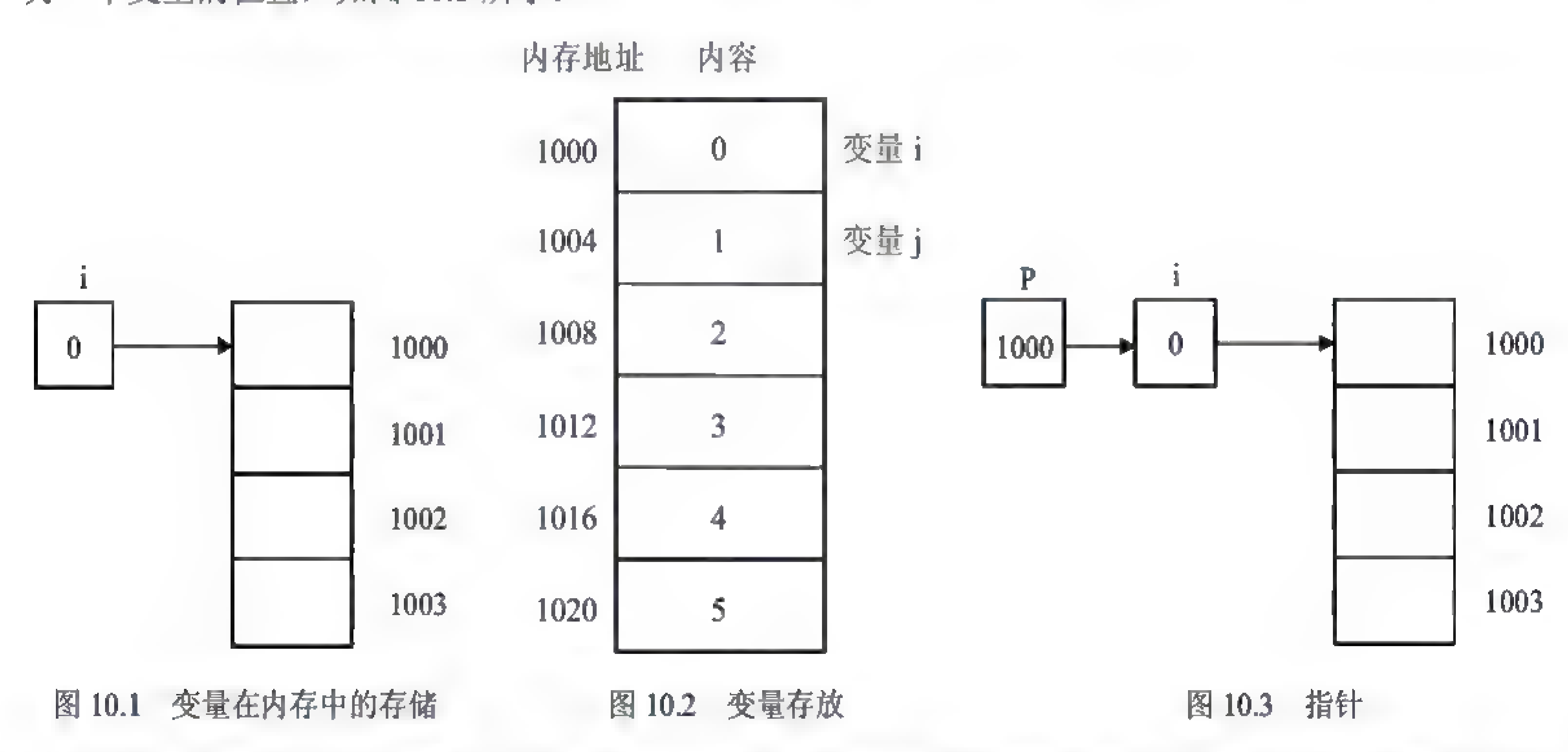


图10.1 变量在内存中的存储　　图10.2 变量存放　　图10.3 指针

在程序中定义了一个变量，在进行编译时就会给该变量在内存中分配一个地址，通过访问这个地址可以找到所需的变量，这个变量的地址称为该变量的"指针"。图10.3所示的地址1000是变量i的指针。

10.1.2 变量与指针

变量的地址是变量和指针二者之间连接的纽带，如果一个变量包含了另一个变量的地址，则可以理解成第一个变量指向第二个变量。所谓"指向"就是通过地址来体现的。因为指针变量是指向一个

变量的地址，所以将一个变量的地址值赋给这个指针变量后，这个指针变量就“指向”了该变量。例如，将变量 i 的地址存放到指针变量 p 中，p 就指向 i，其关系如图 10.4 所示。

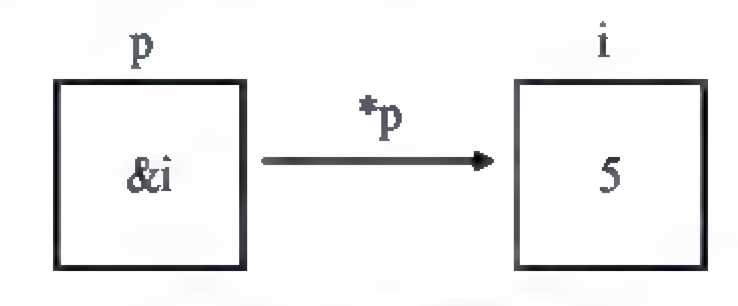


图 10.4　地址与指针

在程序代码中是通过变量名对内存单元进行存取操作的，但是代码经过编译后已经将变量名转换为该变量在内存中的存放地址，对变量值的存取都是通过地址进行的。如对图 10.2 所示的变量 i 和变量 j 进行如下操作：

```
i+j;
```

其含义是：根据变量名与地址的对应关系，找到变量 i 的地址 1000，然后从 1000 开始读取 4 个字节数据放到 CPU 寄存器中，再找到变量 j 的地址 1004，从 1004 开始读取 4 个字节的数据放到 CPU 的另一个寄存器中，通过 CPU 的加法中断计算出结果。

在低级语言的汇编语言中都是直接通过地址来访问内存单元的，在高级语言中一般使用变量名访问内存单元，但 C 语言作为高级语言提供了通过地址来访问内存单元的方式。

10.1.3　指针变量

由于通过地址能访问指定的内存存储单元，可以说地址“指向”该内存单元。地址可以形象地称为指针，意思是通过指针能找到内存单元。一个变量的地址称为该变量的指针。如果有一个变量专门用来存放另一个变量的地址，它就是指针变量。在 C 语言中有专门用来存放内存单元地址的变量类型，即指针类型。下面将针对如何定义一个指针变量、如何为一个指针变量赋值及如何引用指针变量这 3 个方面加以介绍。

1. 指针变量的一般形式

如果有一个变量专门用来存放另一变量的地址，则它称为指针变量。图 10.4 所示的 p 就是一个指针变量。如果一个变量包含指针（指针等同于一个变量的地址），则必须对它进行说明。定义指针变量的一般形式如下：

```
类型说明 * 变量名
```

其中，“*”表示该变量是一个指针变量，变量名即为定义的指针变量名，类型说明表示本指针变量所指向的变量的数据类型。

2. 指针变量的赋值

指针变量同普通变量一样，使用之前不仅需要定义，而且必须赋予具体的值。未经赋值的指针变量不能使用。给指针变量所赋的值与给其他变量所赋的值不同，给指针变量的赋值只能赋予地址，而不能赋予任何其他数据，否则将引起错误。C 语言中提供了地址运算符“&”来表示变量的地址。其一

般形式如下：

```
& 变量名;
```

如&a 表示变量 a 的地址，&b 表示变量 b 的地址。给一个指针变量赋值可以有以下两种方法。

（1）定义指针变量的同时就进行赋值，例如：

```
int a;
int *p=&a;
```

（2）先定义指针变量之后再赋值，例如：

```
int a;
int *p;
p=&a;
```

注意

注意这两种赋值语句的区别，如果在定义完指针变量之后再赋值注意不要加“*”。

【例 10.1】　从键盘中输入两个数，利用指针的方法将这两个数输出。（**实例位置：资源包\TM\sl\10\1**）

```
#include<stdio.h>
main()
{
    int a, b;
    int *ipointer1, *ipointer2;                /*声明两个指针变量*/
    scanf("%d,%d", &a, &b);                    /*输入两个数*/
    ipointer1 = &a;
    ipointer2 = &b;                            /*将地址赋给指针变量*/
    printf("The number is:%d,%d\n", *ipointer1, *ipointer2);
}
```

程序运行结果如图 10.5 所示。

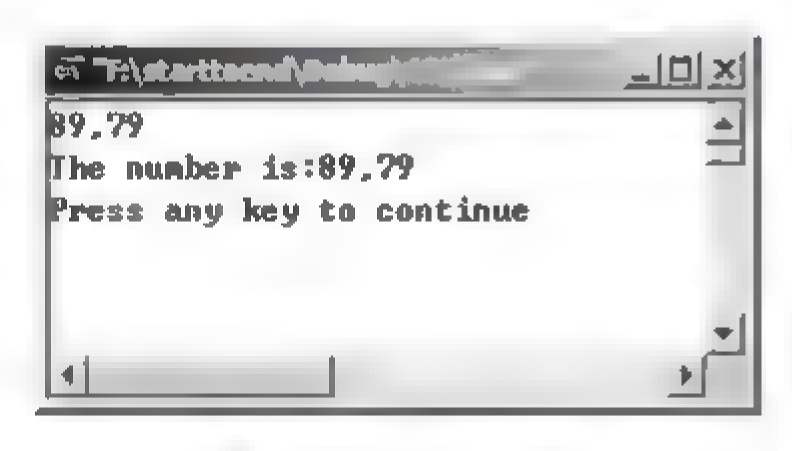


图 10.5　数据输出

通过例 10.1 可以发现，程序中采用的是第二种赋值方式，即先定义再赋值。

这里强调一点，即不允许把一个数赋予指针变量，例如：

```
int *p;
p=1002;
```

这样写是错误的。

3. 指针变量的引用

引用指针变量是对变量进行间接访问的一种形式。对指针变量的引用形式如下：

```
*指针变量
```

其含义是引用指针变量所指向的值。

【例 10.2】 利用指针变量实现数据的输入和输出。（**实例位置：资源包\TM\sl\10\2**）

```
#include<stdio.h>
main()
{
	int *p,q;
	printf("please input:\n");
	scanf("%d",&q);                                    /*输入一个整型数据*/
	p = &q;
	printf("the number is:\n");
	printf("%d\n",*p);                                 /*输出变量的值*/
}
```

程序运行结果如图 10.6 所示。

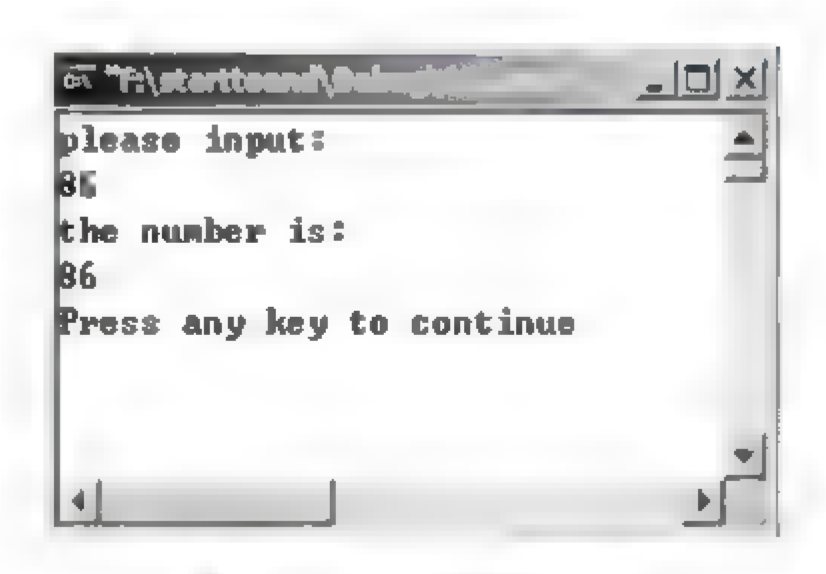


图 10.6 指针变量应用

可将上述程序修改成如下形式：

```
#include<stdio.h>
main()
{
	int *p,q;
	p=&q;
	printf("please input:\n");                         /*输出变量的地址*/
	scanf("%d",p);
	printf("the number is:\n");
	printf("%d\n",q);                                  /*输出变量的值*/
}
```

运行结果完全相同。

4. “&” 和 “*” 运算符

在前面介绍指针变量的过程中，用到了“&”和“*”两个运算符，运算符“&”是一个返回操作

数地址的单目运算符，叫作取地址运算符，例如：

```
p=&i;
```

就是将变量 i 的内存地址赋给 p，这个地址是该变量在计算机内部的存储位置。

运算符“*”是单目运算符，叫作指针运算符，作用是返回指定地址内变量的值。如前面提到过 p 中装有变量 i 的内存地址，则：

```
q=*p;
```

就是将变量 i 的值赋给 q，假如变量 i 的值是 5，则 q 的值也是 5。

5. “&*”和“*&”的区别

如果有如下语句：

```
int a;
p=&a;
```

下面通过以上两条语句来分析“&”和“*&”的区别，“&”和“*”的运算符优先级别相同，按自右而左的方向结合。因此“&*p”先进行“*”运算，“*p”相当于变量 a；再进行“&”运算，“&*p”就相当于取变量 a 的地址。“*&a”先进行“&”运算，“&a”就是取变量 a 的地址，然后执行“*”运算，“*&a”就相当于取变量 a 所在地址的值，实际就是变量 a。下面通过两个实例来具体介绍。

【例 10.3】 “&*”的应用。（**实例位置：资源包\TM\sl\10\3**）

```
#include<stdio.h>
main()
{
    long i;
    long *p;
    printf("please input the number:\n");
    scanf("%ld",&i);
    p=&i;
    printf("the result1 is: %ld\n",&*p);                /*输出变量 i 的地址*/
    printf("the result2 is: %ld\n",&i);                 /*输出变量 I 的地址*/
}
```

程序运行结果如图 10.7 所示。

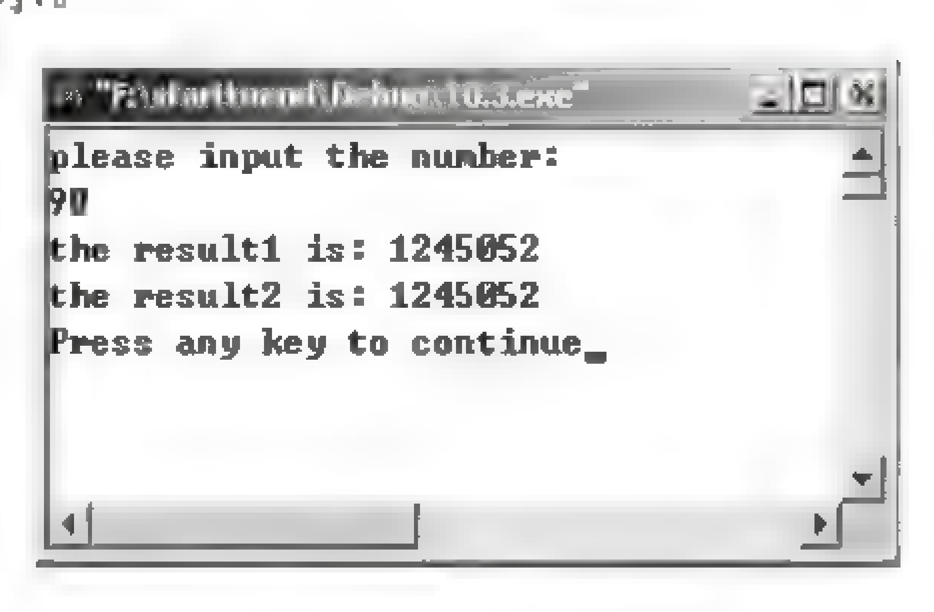


图 10.7 “&*”的应用

【例 10.4】 “*&”的应用。（**实例位置：资源包\TM\sl\10\4**）

```
#include<stdio.h>
main()
{
    long i;
    long *p;
    printf("please input the number:\n");
    scanf("%ld",&i);
    p=&i;
    printf("the result1 is: %ld\n",*&i);                    /*输出变量 i 的地址*/
    printf("the result2 is: %ld\n",i);                      /*输出变量 i 的值*/
    printf("the result3 is: %ld\n",*p);                     /*使用指针形式输出 i 的地址*/
}
```

程序运行结果如图 10.8 所示。

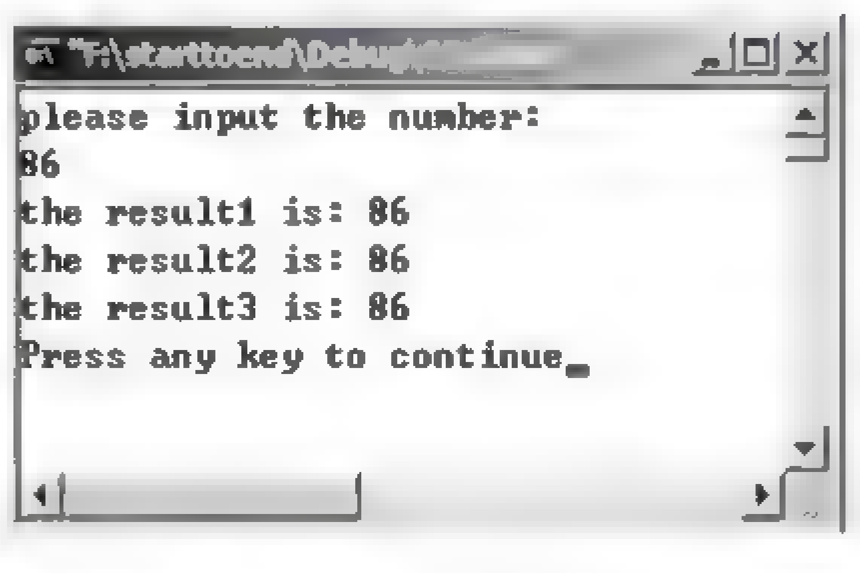


图 10.8 “*&”的应用

10.1.4 指针自加自减运算

指针的自加自减运算不同于普通变量的自加自减运算，也就是说，并非简单地加 1 减 1。这里通过下面的实例进行具体分析。

【例 10.5】 整型变量地址输出。（**实例位置：资源包\TM\sl\10\5**）

```
#include<stdio.h>
main()
{
    int i;
    int *p;
    printf("please input the number:\n");
    scanf("%d",&i);
    p=&i;                                                   /*将变量 i 的地址赋给指针变量*/
    printf("the result1 is: %d\n",p);
    p++;                                                    /*地址加 1，这里的 1 并不代表一个字节*/
    printf("the result2 is: %d\n",p);
}
```

程序运行结果如图 10.9 所示。

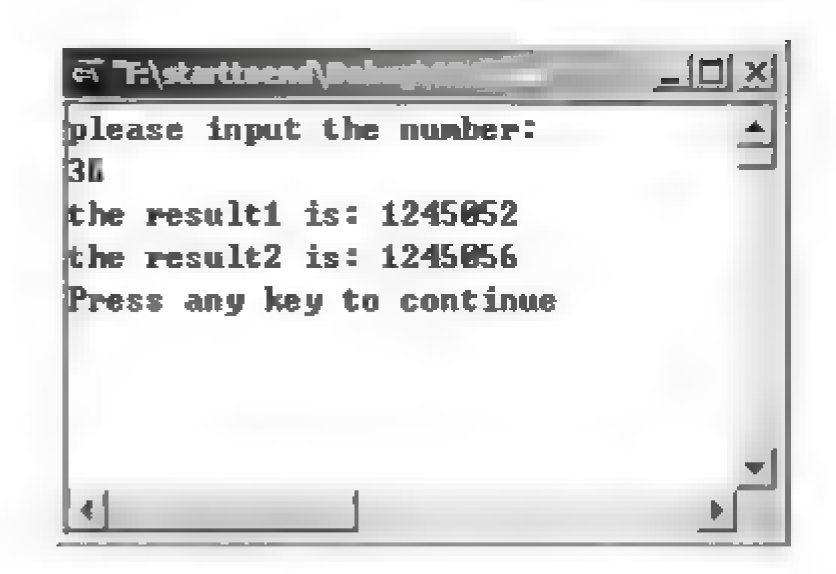


图 10.9　整型变量地址输出

若将例 10.5 改成：

```
#include<stdio.h>
main()
{
    short i;
    short *p;
    printf("please input the number:\n");
    scanf("%d",&i);
    p=&i;                                    /*将变量 i 的地址赋给指针变量*/
    printf("the result1 is: %d\n",p);
    p++;                                     /*地址加 1，这里的 1 并不代表一个字节*/
    printf("the result2 is: %d\n",p);
}
```

程序运行结果如图 10.10 所示。

基本整型变量 i 在内存中占 4 个字节，指针 p 是指向变量 i 的地址的，这里的 p++不是简单地在地址上加 1，而是指向下一个存放基本整型数的地址。图 10.9 所示的结果是因为变量 i 是基本整型，所以执行 p++后，p 的值增加 4（4 个字节）；图 10.10 所示的结果是因为 i 被定义成了短整型，所以执行 p++后，p 的值增加了 2（两个字节）。

指针都按照它所指向的数据类型的直接长度进行增或减。可以将例 10.5 用图 10.11 来形象地表示。

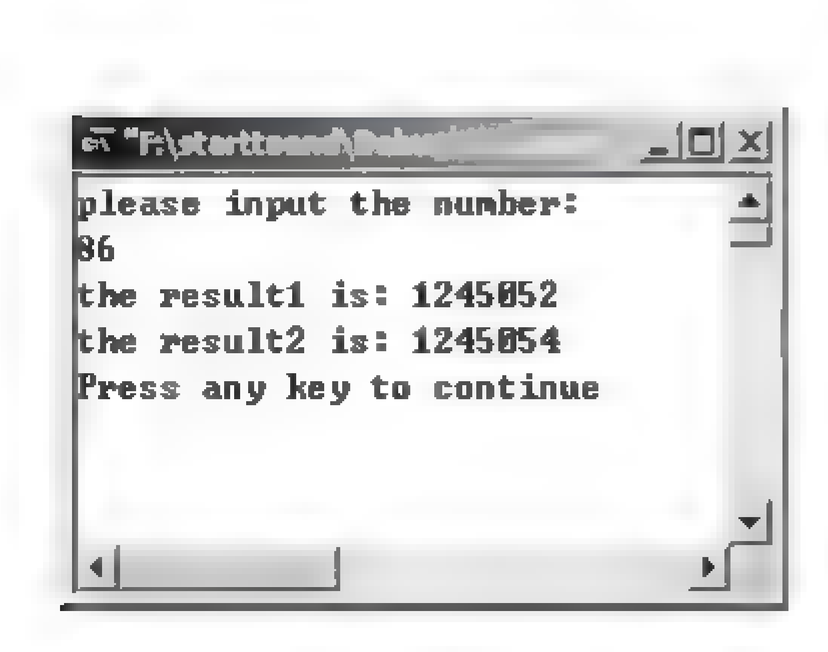


图 10.10　短整型变量地址输出

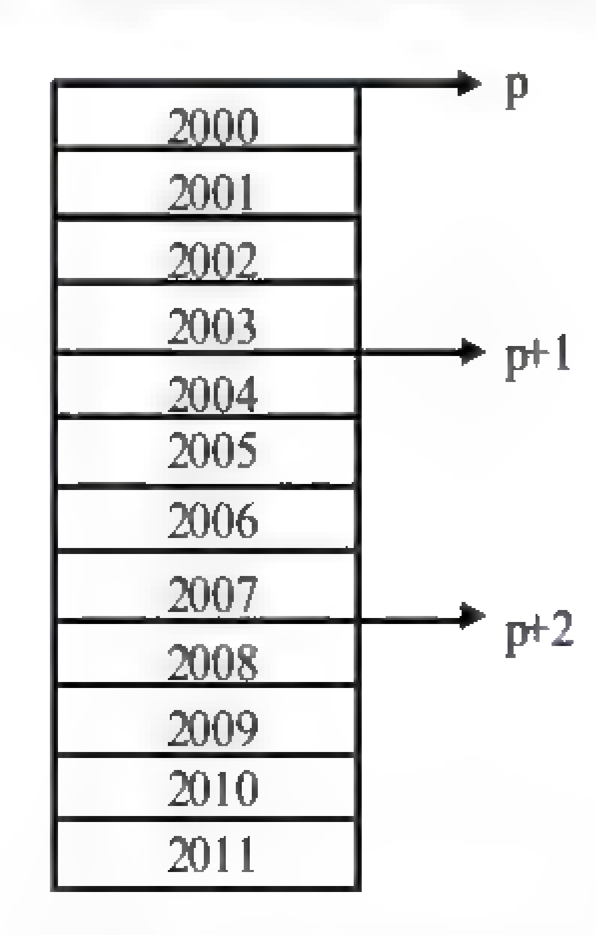


图 10.11　指向整型变量的指针



10.2 数组与指针

系统需要提供一段连续的内存来存储数组中的各元素，内存都有地址，指针变量就是存放地址的变量，如果把数组的地址赋给指针变量，就可以通过指针变量来引用数组。下面就介绍如何用指针来引用一维数组及二维数组元素。

10.2.1 一维数组与指针

当定义一个一维数组时，系统会在内存中为该数组分配一个存储空间，其数组的名称就是数组在内存中的首地址。若再定义一个指针变量，并将数组的首地址传给指针变量，则该指针就指向了这个一维数组。

例如：

```
int *p,a[10];
p=a;
```

这里 a 是数组名，也就是数组的首地址，将它赋给指针变量 p，也就是将数组 a 的首地址赋给 p。也可以写成如下形式：

```
int *p,a[10];
p=&a[0];
```

上面的语句是将数组 a 中的首个元素的地址赋给指针变量 p。由于 a[0]的地址就是数组的首地址，因此两条赋值操作效果完全相同，如例 10.6 所示。

【例 10.6】 输出数组中的元素。（**实例位置：资源包\TM\sl\10\6**）

```
#include<stdio.h>
main()
{
    int *p,*q,a[5],b[5],i;
    p=&a[0];
    q=b;
    printf("please input array a:\n");
    for(i=0;i<5;i++)
        scanf("%d",&a[i]);
    printf("please input array b:\n");
    for(i=0;i<5;i++)
        scanf("%d",&b[i]);
    printf("array a is:\n");
    for(i=0;i<5;i++)
        printf("%5d",*(p+i));
```

```
	printf("\n");
	printf("array b is:\n");
	for(i=0;i<5;i++)
		printf("%5d",*(q+i));
	printf("\n");
}
```

程序运行结果如图 10.12 所示。

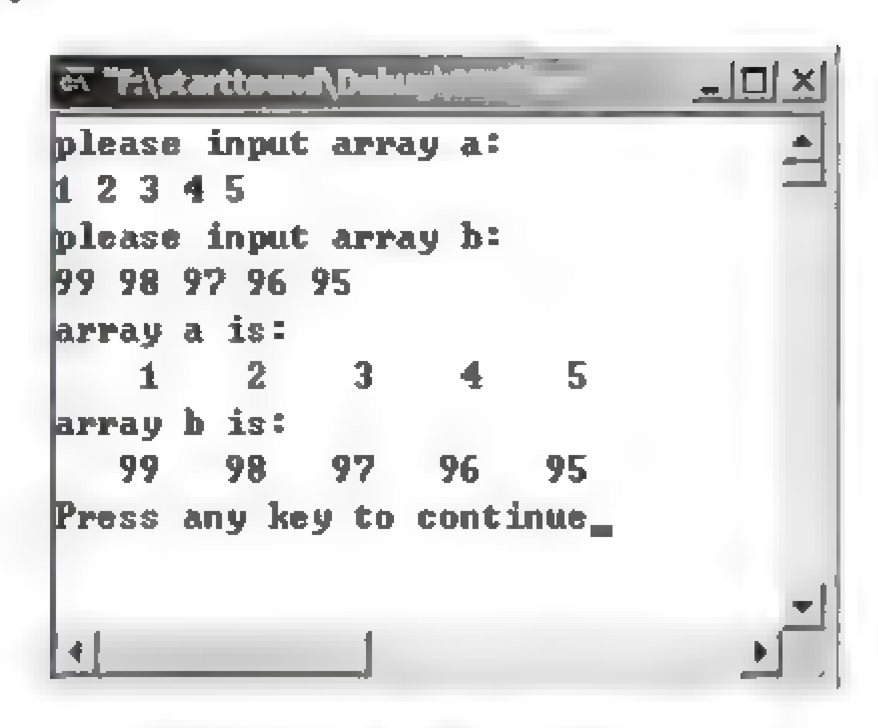


图 10.12　输出数组中的元素

例 10.6 中有如下两条语句：

```
p=&a[0];
q=b;
```

这两种表示方法都是将数组首地址赋给指针变量。

那么如何通过指针的方式来引用一维数组中的元素呢？有以下语句：

```
int *p,a[5];
p=&a;
```

针对上面的语句将通过以下几方面进行介绍：

☑ p+n 与 a+n 表示数组元素 a[n]的地址，即&a[n]。对整个 a 数组来说，共有 5 个元素，n 的取值为 0～4，则数组元素的地址就可以表示为 p+0～p+4 或 a+0～a+4。

☑ 表示数组中的元素用到了前面介绍的数组元素的地址，用*(p+n)和*(a+n)来表示数组中的各元素。

例 10.6 中的语句：

```
printf("%5d",*(p+i));
```

和语句：

```
printf("%5d",*(q+i));
```

分别表示输出数组 a 和数组 b 中对应的元素。

例 10.6 中使用指针指向一维数组及通过指针引用数组元素的过程可以通过图 10.13 和图 10.14 来表示。

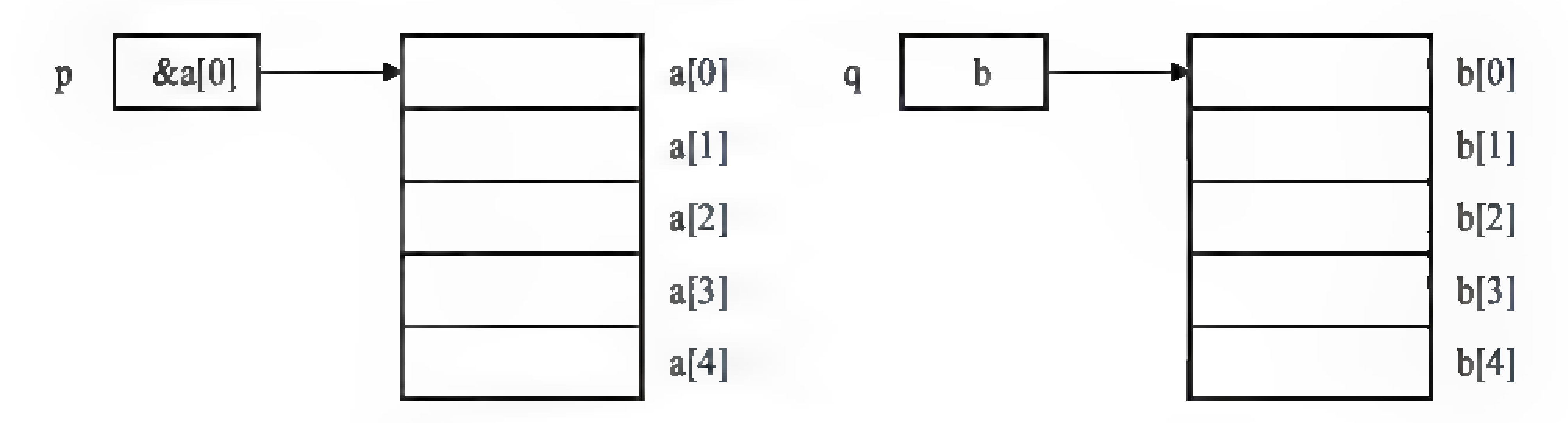


图 10.13　指针指向一维数组

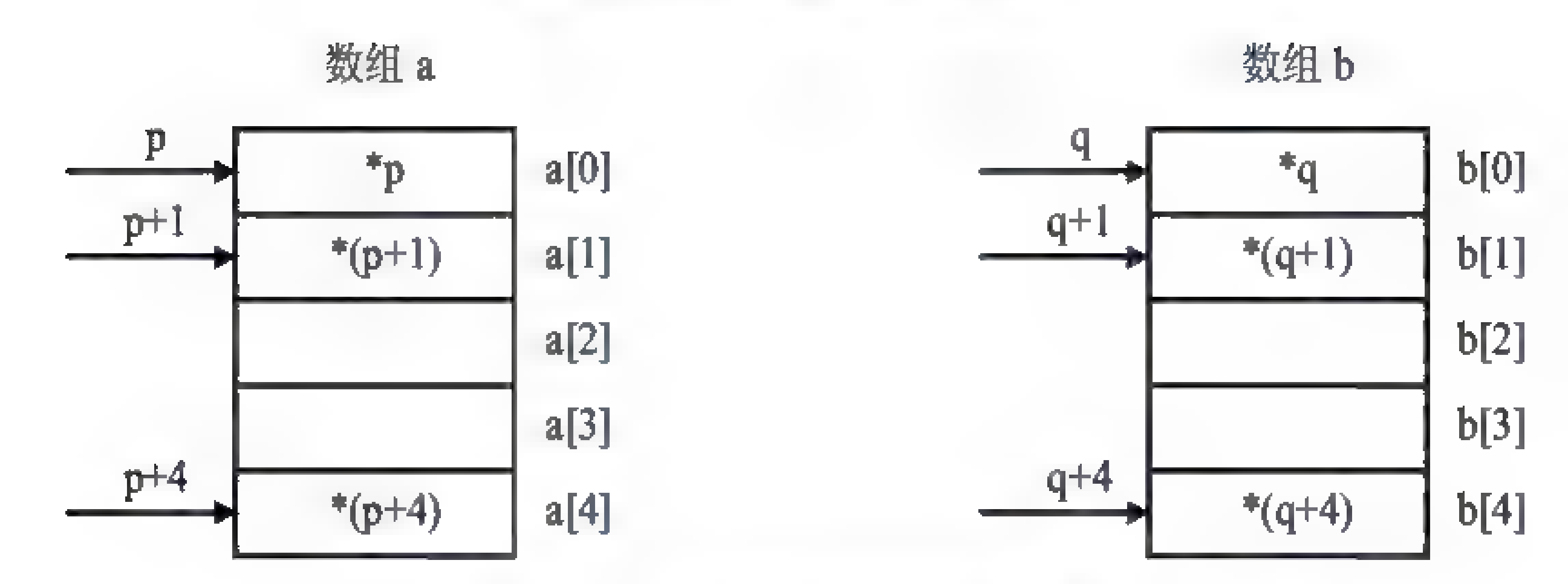


图 10.14　通过指针引用数组元素

前面提到，可以用 a+n 表示数组元素的地址，*(a+n)表示数组元素，那么就可以将例 10.6 的程序代码改成如下形式：

```
#include<stdio.h>
main()
{
	int *p,*q,a[5],b[5],i;
	p=&a[0];
	q=b;
	printf("please input array a:\n");
	for(i=0;i<5;i++)
		scanf("%d",&a[i]);
	printf("please input array b:\n");
	for(i=0;i<5;i++)
		scanf("%d",&b[i]);
	printf("array a is:\n");
	for(i=0;i<5;i++)
		printf("%5d",*(a+i));
	printf("\n");
	printf("array b is:\n");
	for(i=0;i<5;i++)
		printf("%5d",*(b+i));
	printf("\n");
}
```

程序运行的结果与例 10.6 一样。

☑　表示指针的移动可以使用“++”和“--”这两个运算符。

利用“++”运算符可将程序改写成如下形式：

```
#include<stdio.h>
main()
{
    int *p,*q,a[5],b[5],i;
    p=&a[0];
    q=b;
    printf("please input array a:\n");
    for(i=0;i<5;i++)
        scanf("%d",&a[i]);
    printf("please input array b:\n");
    for(i=0;i<5;i++)
        scanf("%d",&b[i]);
    printf("array a is:\n");
    for(i=0;i<5;i++)
        printf("%5d",*p++);
    printf("\n");
    printf("array b is:\n");
    for(i=0;i<5;i++)
        printf("%5d",*q++);
    printf("\n");
}
```

还可将上面程序再进一步改写，运行结果仍与例 10.6 相同。改写后的程序代码如下：

```
#include<stdio.h>
main()
{
    int *p,*q,a[5],b[5],i;
    p=&a[0];
    q=b;
    printf("please input array a:\n");
    for(i=0;i<5;i++)
        scanf("%d",p++);
    printf("please input array b:\n");
    for(i=0;i<5;i++)
        scanf("%d",q++);
    p=a;
    q=b;
    printf("array a is:\n");
    for(i=0;i<5;i++)
        printf("%5d",*p++);
    printf("\n");
    printf("array b is:\n");
    for(i=0;i<5;i++)
        printf("%5d",*q++);
    printf("\n");
}
```

比较上面两个程序会发现，如果在给数组元素赋值时使用了如下语句：

```
printf("please input array a:\n");
for(i=0;i<5;i++)
    scanf("%d",p++);
printf("please input array b:\n");
for(i=0;i<5;i++)
    scanf("%d",q++);
```

而且在输出数组元素时需要使用指针变量，则需要加上如下语句：

```
p=a;
q=b;
```

这两个语句的作用是将指针变量 p 和 q 重新指向数组 a 和数组 b 在内存中的起始位置。若没有该语句，而直接使用*p++的方法进行输出，将会产生错误。

10.2.2 二维数组与指针

定义一个 3 行 5 列的二维数组，其在内存中的存储形式如图 10.15 所示。

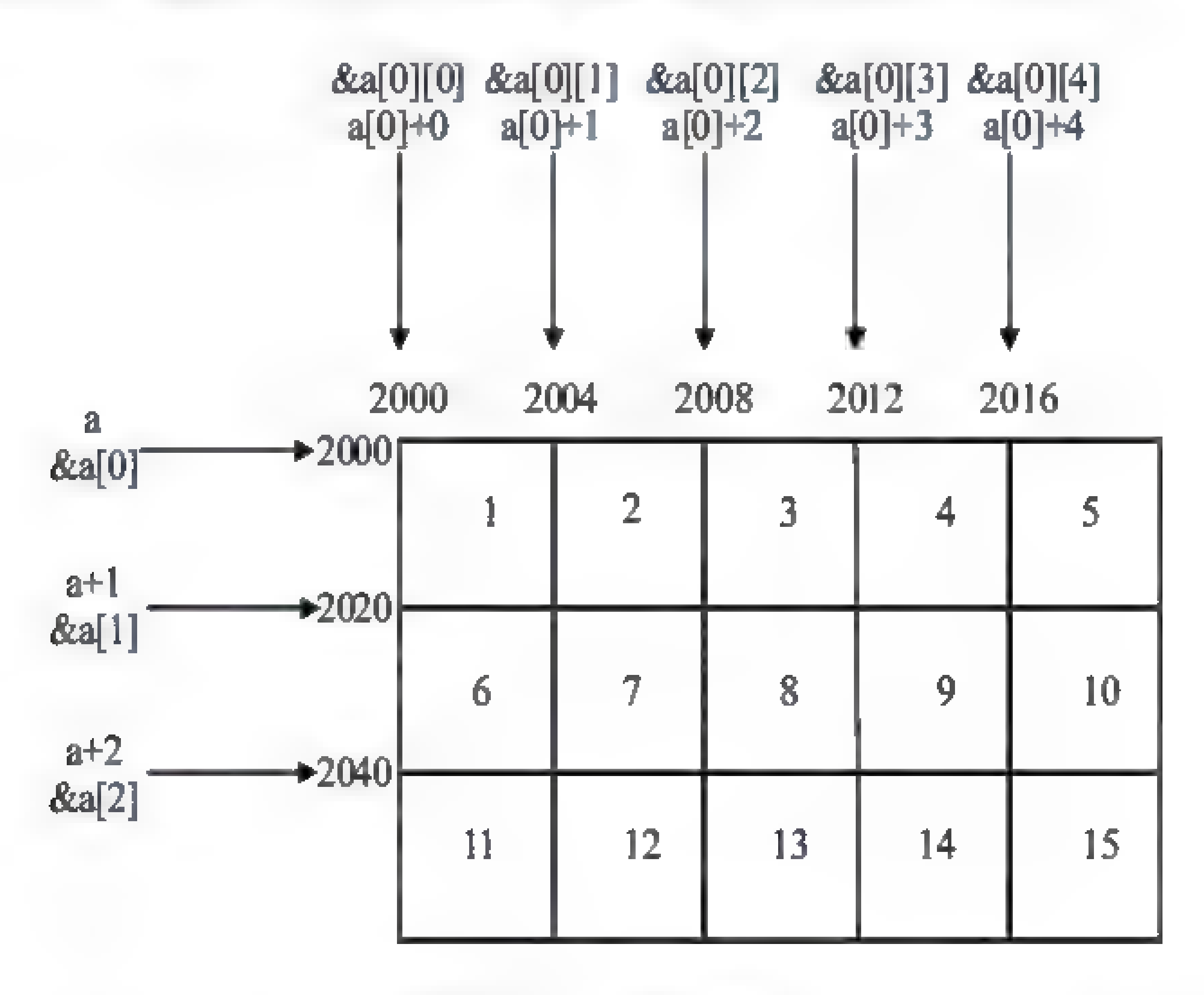


图 10.15 二维数组

从图 10.15 中可以看到几种表示二维数组中元素地址的方法，下面逐一进行介绍。

☑ &a[0][0]既可以看作数组 0 行 0 列的首地址，也可以看作二维数组的首地址。&a[m][n]就是第 m 行 n 列元素的地址。

☑ a[0]+n 表示第 0 行第 n 个元素的地址。

【例 10.7】 利用指针对二维数组进行输入和输出。（**实例位置：资源包\TM\sl\10\7**）

```
#include<stdio.h>
main()
{
```

```
    int a[3][5],i,j;
    printf("please input:\n");
    for(i=0;i<3;i++)                              /*控制二维数组的行数*/
    {
        for(j=0;j<5;j++)                          /*控制二维数组的列数*/
        {
            scanf("%d",a[i]+j);                   /*给二维数组元素赋初值*/
        }
    }
    printf("the array is:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<5;j++)
        {
            printf("%5d",*(a[i]+j));              /*输出数组中的元素*/
        }
        printf("\n");
    }
}
```

程序运行结果如图 10.16 所示。

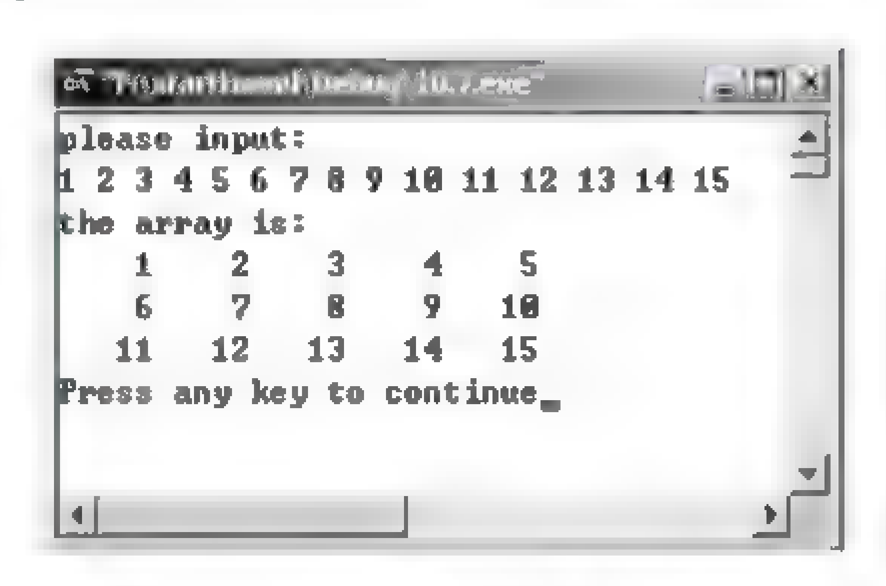


图 10.16　二维数组的输入和输出

在运行结果仍相同的前提下，还可将程序改写成如下形式：

```
#include<stdio.h>
main()
{
    int a[3][5],i,j,*p;
    p=a[0];
    printf("please input:\n");
    for(i=0;i<3;i++)                              /*控制二维数组的行数*/
    {
        for(j=0;j<5;j++)                          /*控制二维数组的列数*/
        {
            scanf("%d",p++);                      /*为二维数组中的元素赋值*/
        }
    }
    p=a[0];                                       /*p 为第一个元素的地址*/
    printf("the array is:\n");
    for(i=0;i<3;i++)
    {
```

```
            for(j=0;j<5;j++)
            {
                printf("%5d",*p++);                    /*输出二维数组中的元素*/
            }
            printf("\n");
        }
}
```

☑ &a[0]是第 0 行的首地址，当然&a[n]就是第 n 行的首地址。

【例 10.8】 将一个 3 行 5 列二维数组的第 3 行元素输出。（**实例位置：资源包\TM\sl\10\8**）

```
#include<stdio.h>
main()
{
    int a[3][5],i,j,(*p)[5];
    p=&a[0];
    printf("please input:\n");
    for(i=0;i<3;i++)                                   /*控制二维数组的行数*/
        for(j=0;j<5;j++)                               /*控制二维数组的列数*/
            scanf("%d",(*(p+i))+j);                    /*为二维数组中的元素赋值*/
        p=&a[2];                                       /*p 为第一个元素的地址*/
        printf("the third line is:\n");
            for(j=0;j<5;j++)
                printf("%5d",*((*p)+j));               /*输出二维数组中的元素*/
            printf("\n");
}
```

程序运行结果如图 10.17 所示。

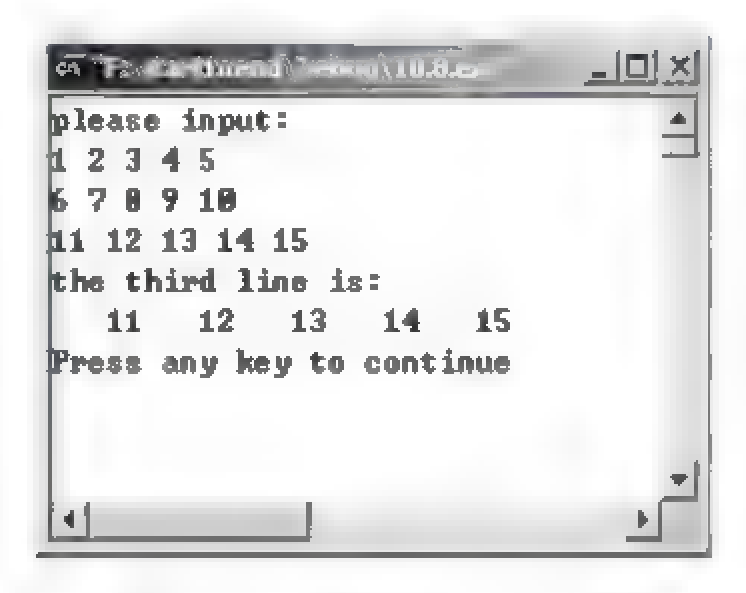


图 10.17 输出第 3 行元素

☑ a+n 表示第 n 行的首地址。

【例 10.9】 将一个 3 行 5 列的二维数组的第 2 行元素输出。（**实例位置：资源包\TM\sl\10\9**）

```
#include<stdio.h>
main()
{
    int a[3][5],i,j;
    printf("please input:\n");
    for(i=0;i<3;i++)                                   /*控制二维数组的行数*/
        for(j=0;j<5;j++)                               /*控制二维数组的列数*/
            scanf("%d",*(a+i)+j);                      /*为二维数组中的元素赋值*/
            /*p 为第一个元素的地址*/
```

```
        printf("the second line is:\n");
        for(j=0;j<5;j++)
                printf("%5d",*(*(a+1)+j));                    /*输出二维数组中的元素*/
            printf("\n");
}
```

程序运行结果如图 10.18 所示。

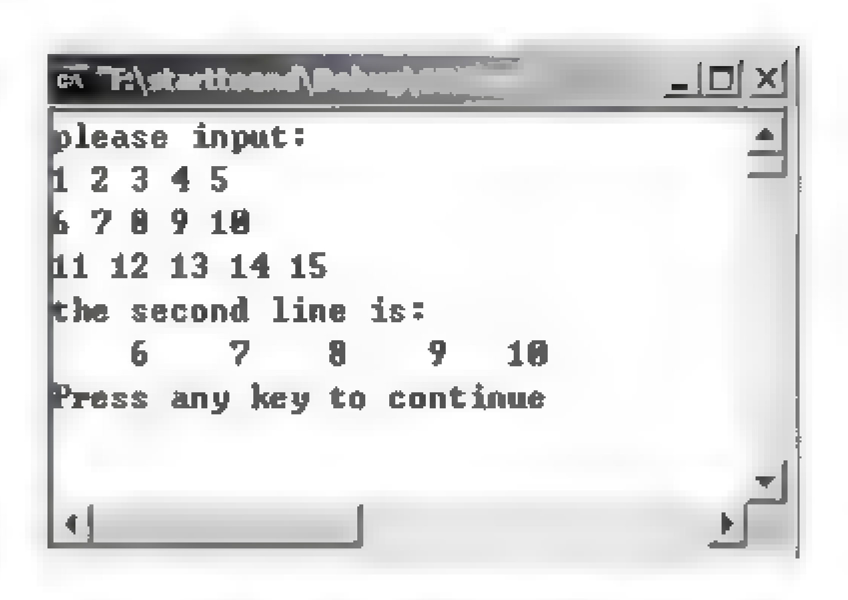
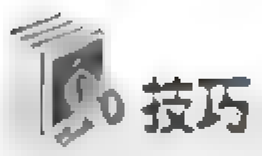


图 10.18　输出第 2 行元素

前面讲过了如何利用指针来引用一维数组，这里在一维数组的基础上介绍如何通过指针来引用一个二维数组中的元素。

☑　*(*(a+n)+m)表示第 n 行第 m 列元素。

☑　*(a[n]+m)表示第 n 行第 m 列元素。



技巧

利用指针引用二维数组的关键是要记住*(a+i)与 a[i]是等价的。

10.2.3　字符串与指针

可以通过两种方式访问一个字符串，第一种方式就是前面讲过的使用字符数组来存放一个字符串，从而实现对字符串的操作；另一种方式就是下面将要介绍的使用字符指针指向一个字符串，此时可不定义数组。

【例 10.10】　字符型指针应用。(**实例位置：资源包\TM\sl\10\10**)

```
#include<stdio.h>
main()
{
    char *string="hello mingri";
    printf("%s",string);                                  /*输出字符串*/
}
```

程序运行结果如图 10.19 所示。

例 10.10 中定义了字符型指针变量 string，用字符串常量“hello mingri”为其赋初值。注意，这里并不是把“hello mingri”中的所有字符存放到 string 中，只是把该字符串中的第一个字符的地址赋给指针变量 string，如图 10.20 所示。

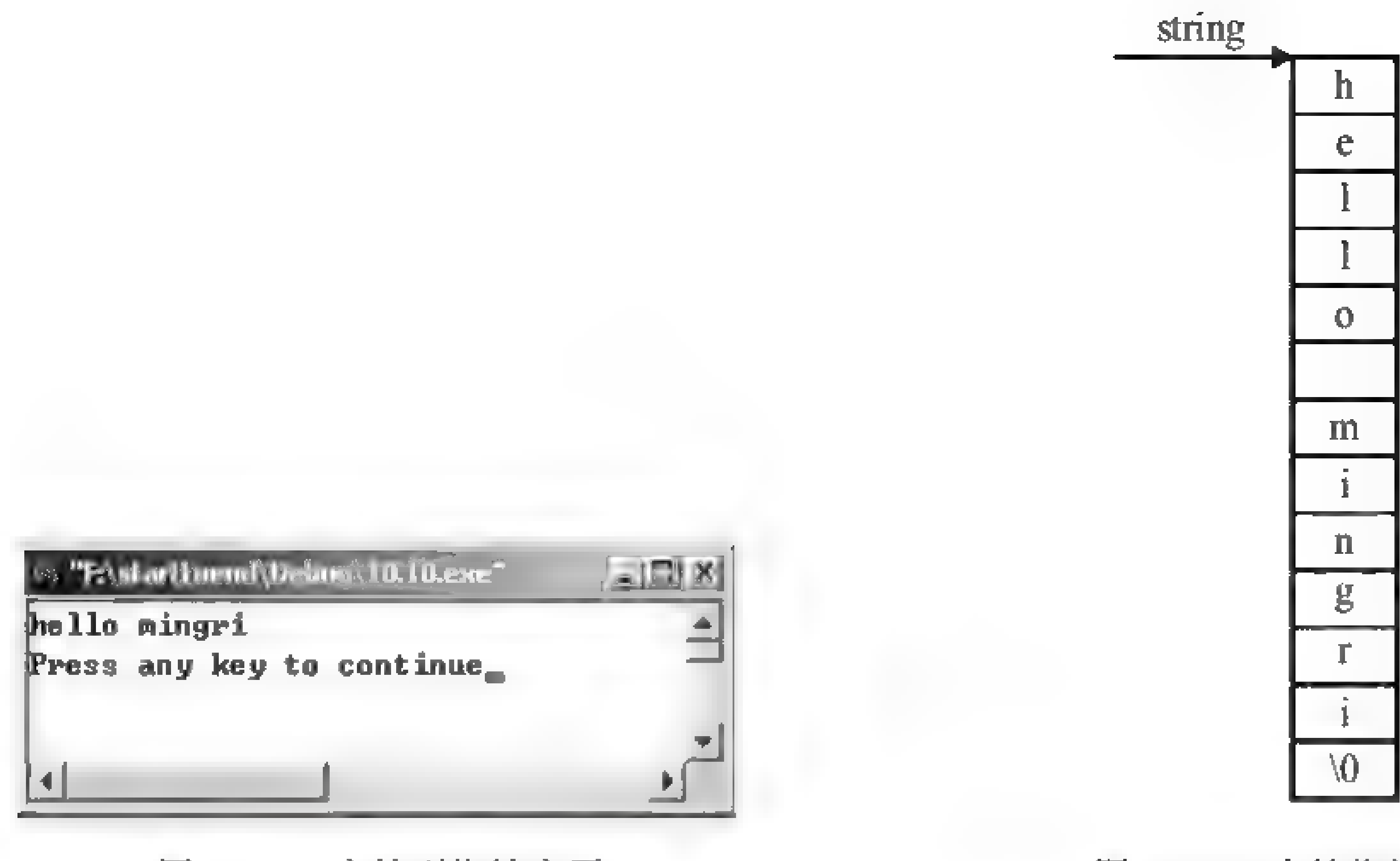


图 10.19　字符型指针应用　　　　图 10.20　字符指针

语句：

```
char *string="hello mingri";
```

等价于下面两条语句：

```
char *string;
string="hello mingri";
```

【例 10.11】　输入两个字符串 a 和 b，将字符串 a 和 b 连接起来。（**实例位置：资源包\TM\sl\10\11**）

```
#include<stdio.h>
main()
{
    char str1[ ]="you are beautiful",str2[30],*p1,*p2;
    p1=str1;
    p2=str2;
    while(*p1!='\0')
    {
        *p2=*p1;
        p1++;                               /*移动指针*/
        p2++;
    }
    *p2='\0';                               /*在字符串的末尾加结束符*/
    printf("Now the string2 is:\n");
    puts(str1);                             /*输出字符串*/
}
```

程序运行结果如图 10.21 所示。

例 10.11 中定义了两个指向字符型数据的指针变量。首先让 p1 和 p2 分别指向字符串 a 和字符串 b 的第一个字符的地址。将 p1 所指向的内容赋给 p2 所指向的元素，然后 p1 和 p2 分别加 1，指向下一个元素，直到*p1 的值为“\0”为止。

这里有一点需要注意，就是 p1 和 p2 的值是同步变化的，如图 10.22 所示。若 p1 处在 p11 的位置，p2 就处在 p21 的位置；若 p1 处在 p12 的位置，p2 就处在 p22 的位置。

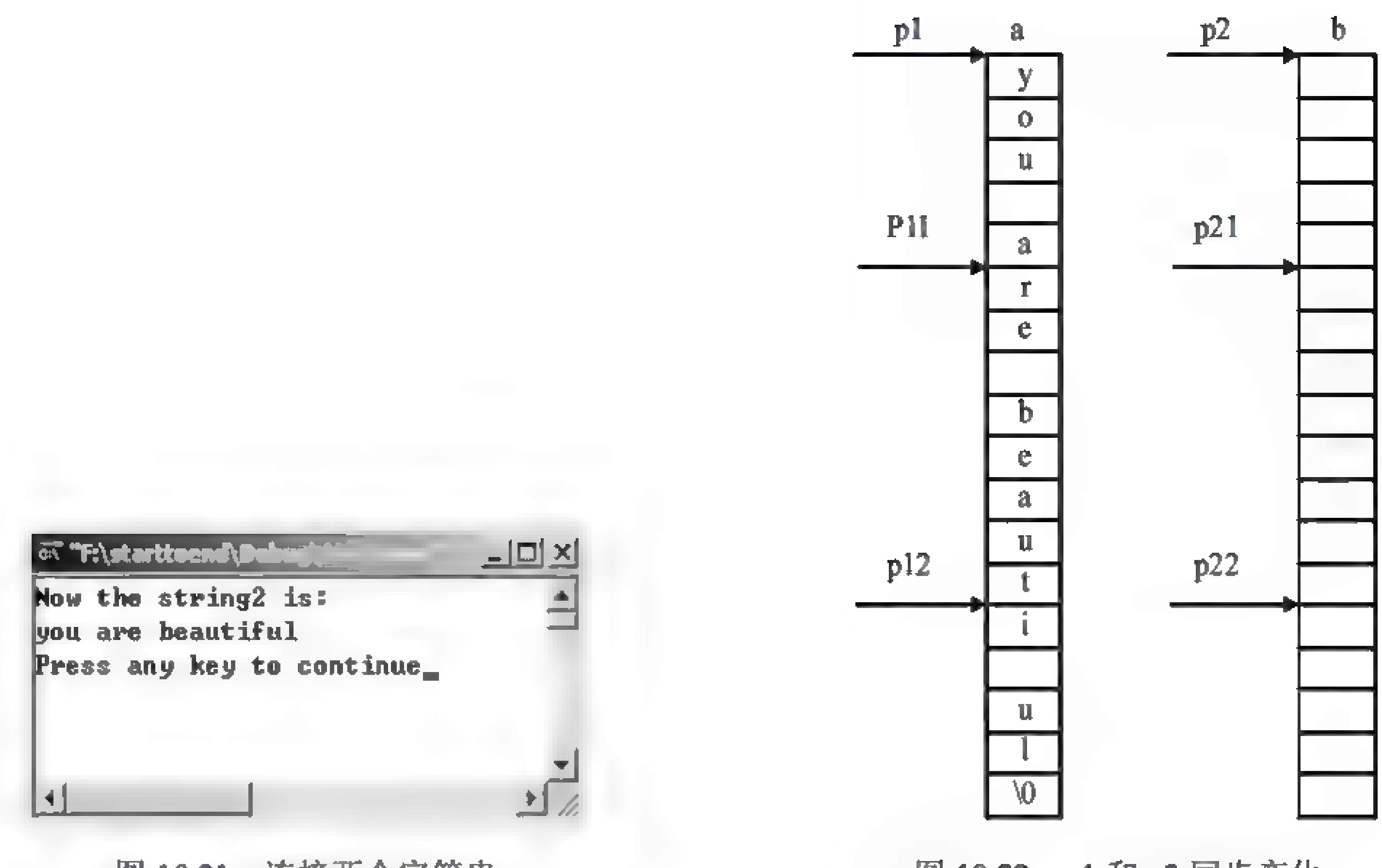


图 10.21　连接两个字符串

图 10.22　p1 和 p2 同步变化

10.2.4　字符串数组

前面讲过了字符数组，这里提到的字符串数组有别于字符数组。字符数组是一个一维数组，而字符串数组是以字符串作为数组元素的数组，可以将其看成一个二维字符数组。下面定义一个简单的字符串数组：

```
char country[5][20]=
{
      "China",
      "Japan",
      "Russia",
      "Germany",
      "Switzerland"
}
```

字符型数组变量 country 被定义为含有 5 个字符串的数组，每个字符串的长度要小于 20（这里要考虑字符串最后的“\0”）。

通过观察上面定义的字符串数组可以发现像"China"和"Japan"这样的字符串的长度仅为 5，加上字符串结束符也仅为 6，而内存中却要给它们分别分配一个 20 字节的空间，这样就会造成资源浪费。为了解决这个问题，可以使用指针数组，使每个指针指向所需要的字符常量，这种方法虽然需要在数组中保存字符指针，而且也占用空间，但要远少于字符串数组需要的空间。

那么什么是指针数组？一个数组，其元素均为指针类型数据，称为指针数组。也就是说，指针数组中的每一个元素都相当于一个指针变量。一维指针数组的定义形式如下：

类型名 数组名[数组长度]

【例 10.12】 输出 12 个月。（**实例位置：资源包\TM\sl\10\12**）

```
#include<stdio.h>
main()
{
	int i;
	char *month[]=
	{
		"January",
		"February",
		"March",
		"April",
		"May",
		"June",
		"July",
		"August",
		"September",
		"October",
		"November",
		"December"
	};												/*给指针数组中的元素赋初值*/
	for(i=0;i<12;i++)
		printf("%s\n",month[i]);							/*输出指针数组中的各元素*/
}
```

程序运行结果如图 10.23 所示。

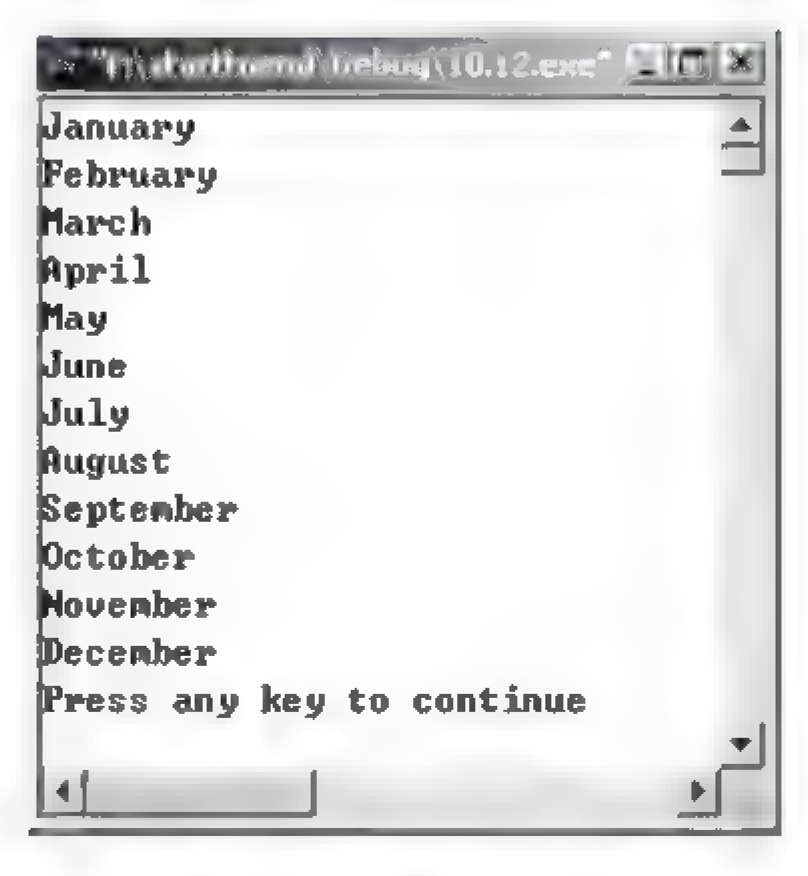


图 10.23 输出 12 个月



10.3 指向指针的指针

一个指针变量可以指向整型变量、实型变量、字符类型变量，当然也可以指向指针类型变量。当这种指针变量用于指向指针类型变量时，则称之为指向指针的指针变量。这种双重指针如图 10.24 所示。

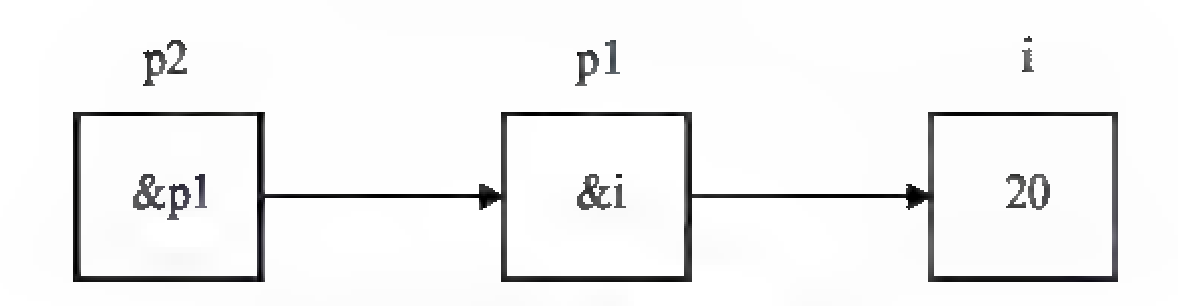


图 10.24　指向指针的指针（一）

整型变量 i 的地址是&i，将其值传递给指针变量 p1，则 p1 指向 i；同时，将 p1 的地址&p1 传递给 p2，则 p2 指向 p1。这里的 p2 就是前面讲到的指向指针变量的指针变量，即指针的指针。指向指针的指针变量定义如下：

```
类型标识符 **指针变量名;
```

例如：

```
int * *p;
```

其含义为定义一个指针变量 p，它指向另一个指针变量，该指针变量又指向一个基本整型变量。由于指针运算符“*”是自右至左结合，所以上述定义相当于：

```
int *(*p);
```

既然知道了如何定义指向指针的指针，那么可以将图 10.24 用图 10.25 更形象地表示出来。

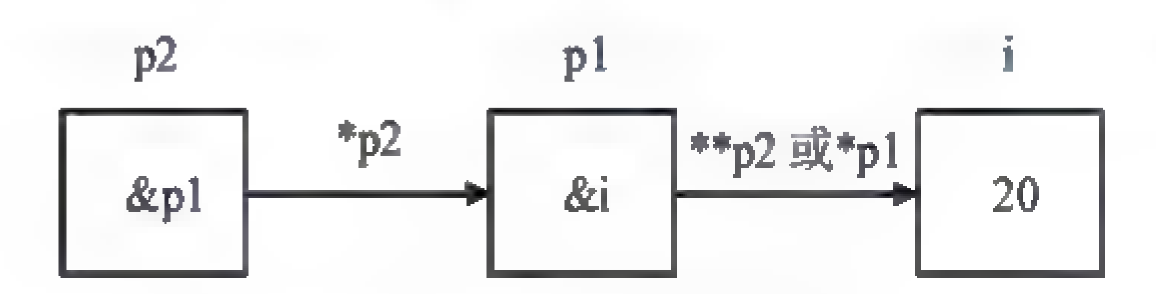


图 10.25　指向指针的指针（二）

下面看一下指向指针的指针变量在程序中是如何应用的。

【例 10.13】　使用指向指针的指针输出 12 个月。（**实例位置：资源包\TM\sl\10\13**）

```
#include<stdio.h>
main()
{
	int i;
	char **p;
	char *month[]=
	{
		"January",
		"February",
		"March",
		"April",
		"May",
		"June",
		"July",
		"August",
		"September",
		"October",
		"November",
```

```
        "December"
    };                                          /*给指针数组中的元素赋初值*/
    for(i=0;i<12;i++)
    {
        p=month+i;
        printf("%s\n",*p);                      /*输出指针数组中的各元素*/
    }
}
```

程序运行结果如图 10.26 所示。



图 10.26　输出 12 个月

【例 10.14】　利用指向指针的指针输出一维数组中是偶数的元素，并统计偶数的个数。（**实例位置：资源包\TM\sl\10\14**）

```
#include<stdio.h>
main()
{
    int a[10],*p1,**p2,i,n=0;                   /*定义数组、指针、变量等为基本整型*/
    printf("please input:\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);                      /*给数组 a 中各元素赋值*/
    p1=a;                                       /*将数组 a 的首地址赋给 p1*/
    p2=&p1;                                     /*将指针 p1 的地址赋给 p2*/
    printf("the array is:");
    for(i=0;i<10;i++)
    {
        if(*(*p2+i)%2==0)
        {
            printf("%5d",*(*p2+i));             /*输出数组中的元素*/
            n++;
        }
    }
    printf("\n");
    printf("the number is:%d\n",n);
}
```

程序运行结果如图 10.27 所示。

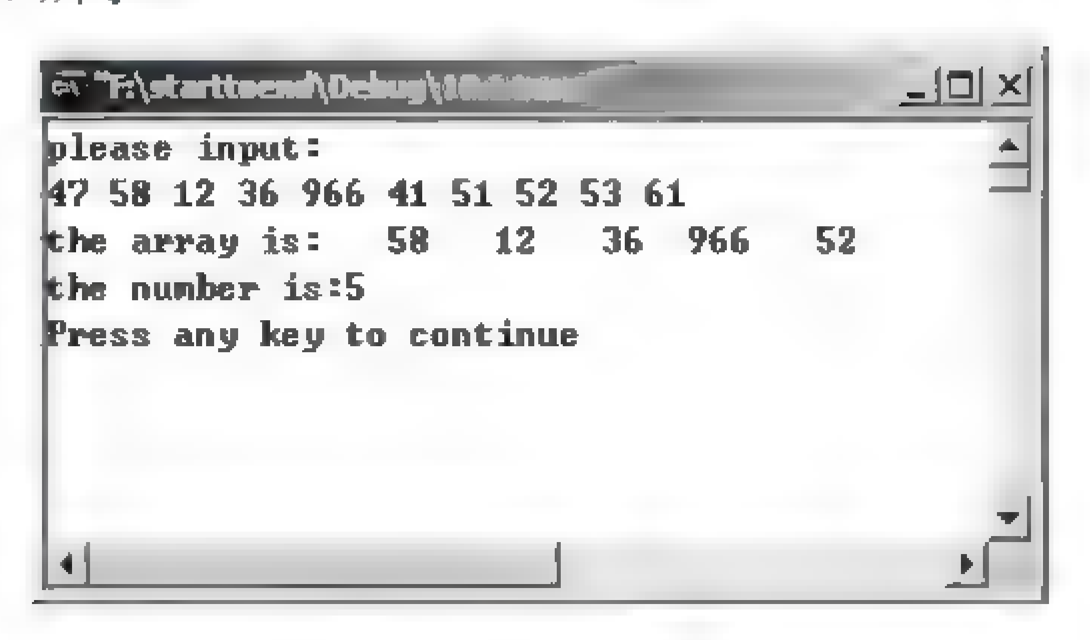


图 10.27 输出偶数

该程序中将数组 a 的首地址赋给指针变量 p1，又将指针变量 p1 的地址赋给 p2，要通过这个双重指针变量 p2 访问数组中的元素，就要一层层地来分析。首先看*p2 的含义，*p2 指向的是指针变量 p1 所存放的内容，即数组 a 的首地址，要想取出数组 a 中的元素，就必须在*p2 前面再加一个指针运算符“*”。

根据前面讲过的指针的用法还可将程序改写成如下形式：

```
#include<stdio.h>
main()
{
    int a[10],*p1,**p2,n=0;                 /*定义数组、指针等为基本整型*/
    printf("please input:\n");
    for(p1=a;p1-a<10;p1++)                  /*指针 p 从 a 的首地址开始变化*/
    {
        p2=&p1;                             /*将指针 p1 的地址赋给 p2*/
        scanf("%d",*p2);                    /*通过指针变量给数组元素赋初值*/
    }
    printf("the array is:");
    for(p1=a;p1-a<10;p1++)
    {
        p2=&p1;                             /*将 p1 地址赋给 p2*/
        if(**p2%2==0)
        {
            printf("%5d",**p2);             /*将数组中的元素输出*/
            n++;
        }
    }
    printf("\n");
    printf("the number is:%d\n",n);
}
```

10.4 指针变量作函数参数



通过前面的介绍可知，整型变量、实型变量、字符型变量、数组名和数组元素等均可作为函数参

数。此外，指针型变量也可以作为函数参数，这里具体进行介绍。

首先通过例 10.15 来介绍如何用指针变量来作函数参数。

【例 10.15】 调用自定义函数交换两个变量的值。（**实例位置：资源包\TM\sl\10\15**）

```
#include <stdio.h>
void swap(int *a,int *b)
{
	int tmp;
	tmp=*a;
	*a=*b;
	*b=tmp;

}
main()
{
	int x,y;
	int *p_x,*p_y;
	printf("请输入两个数：\n");
	scanf("%d",&x);
	scanf("%d",&y);
	p_x=&x;
	p_y=&y;
	swap(p_x,p_y);
	printf("x=%d\n",x);
	printf("y=%d\n",y);
}
```

程序运行结果如图 10.28 所示。



图 10.28 交换两变量的值

swap 函数是用户自定义函数，在 main 函数中调用该函数交换变量 a 和 b 的值，swap 函数的两个形参被传入了两个地址值，也就是传入了两个指针变量。在 swap 函数的函数体内使用整型变量 tmp 作为中间变量，将两个指针变量所指向的数值进行交换。在 main 函数内首先获取输入的两个数值，分别传递给变量 x 和 y，调用 swap 函数将变量 x 和 y 的数值互换。

将前述程序改成如下形式：

```
#include<stdio.h>
void swap(int a,int b)
```

```
{
      int tmp;
      tmp=a;
      a=b;
      b=tmp;

}
void main()
{
      int x,y;
      printf("请输入两个数：\n");
      scanf("%d",&x);
      scanf("%d",&y);
            swap(x,y);
      printf("x=%d\n",x);
      printf("y=%d\n",y);
}
```

程序运行结果如图 10.29 所示。

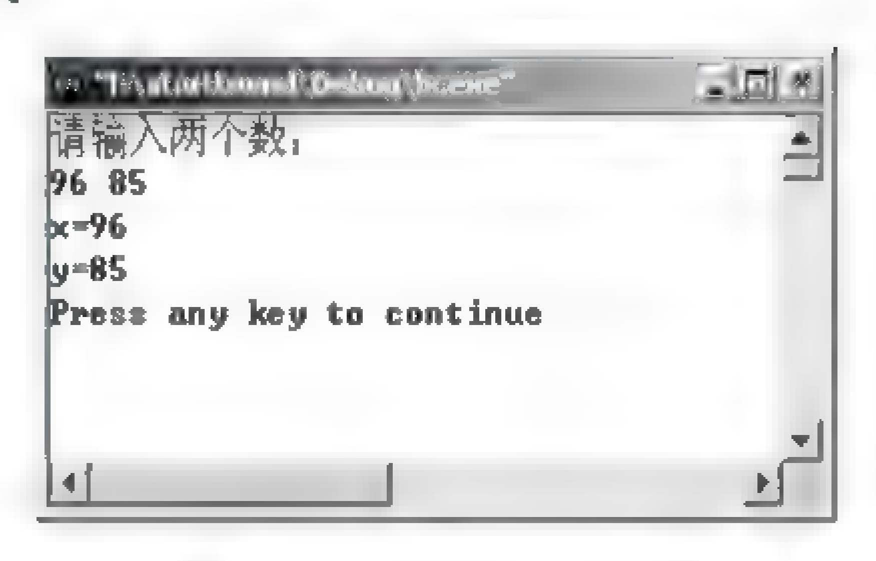


图 10.29　数值未实现交换

程序并没有交换 x 和 y 的值，这涉及值传递概念。

在函数调用过程中，主调用函数与被调用函数之间有一个数值传递过程。

函数调用中发生的数据传递是单向的，只能把实参的值传递给形参，在函数调用过程中，形参的值发生改变，实参的值不会发生变化，因此上面的这段代码同样不能实现 x 和 y 值的互换。

通过指针传递参数可以减少值传递带来的开销，也可以使函数调用不产生值传递。

下面来介绍嵌套的函数调用是如何使用指针变量作函数参数的。

【例 10.16】　嵌套的函数调用。（**实例位置：资源包\TM\sl\10\16**）

```
#include<stdio.h>
void swap(int *p1, int *p2)                              /*自定义交换函数*/
{
     int temp;
     temp =  *p1;
     *p1 =  *p2;
     *p2 = temp;
}
void exchange(int *pt1, int *pt2, int *pt3)              /*3 个数由大到小排序*/
{
```

```
    if (*pt1 <  *pt2)
        swap(pt1, pt2);                              /*调用 swap 函数*/
    if (*pt1 <  *pt3)
        swap(pt1, pt3);
    if (*pt2 <  *pt3)
        swap(pt2, pt3);
}
main()
{
    int a, b, c,  *q1,  *q2,  *q3;
    puts("Please input three key numbers you want to rank:");
    scanf("%d,%d,%d", &a, &b, &c);
    q1 = &a;                                         /*将变量 a 的地址赋给指针变量 q1*/
    q2 = &b;
    q3 = &c;
    exchange(q1, q2, q3);                            /*调用 exchange 函数*/
    printf("\n%d,%d,%d\n", a, b, c);
}
```

程序运行结果如图 10.30 所示。

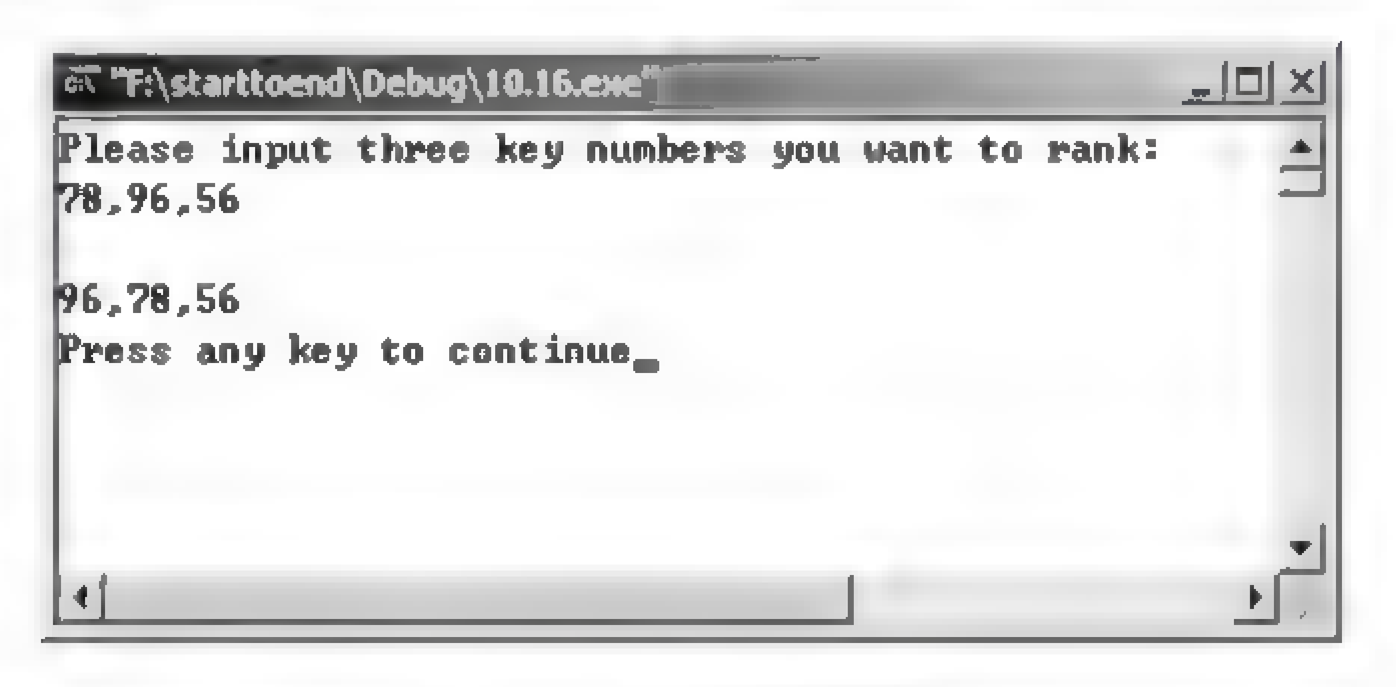


图 10.30　嵌套的函数调用

本程序创建了一个自定义函数 swap，用于交换两个变量的值。本程序还创建了一个 exchange 函数，其作用是将 3 个数由大到小排序，在 exchange 函数中调用了前面自定义的 swap 函数，这里的 swap 和 exchange 函数都是以指针变量作为形参。程序运行时，通过键盘输入 3 个数 a、b、c，分别将 a、b、c 的地址赋给 q1、q2、q3，调用 exchange 函数，将指针变量作为实参，将实参变量的值传递给形参变量，此时 q1 和 pt1 都指向变量 a，q2 和 pt2 都指向变量 b，q3 和 pt3 都指向变量 c；在 exchange 函数中又调用了 swap 函数，当执行 swap(pt1,pt2)时，pt1 也指向了变量 a，pt2 指向了变量 b，这一过程如图 10.31 所示。

C 语言中实参变量和形参变量之间的数据传递是单向的“值传递”方式。指针变量作函数参数也是如此，调用函数不可能改变实参指针变量的值，但可以改变实参指针变量所指变量的值。

前面介绍了指向数组的指针变量的定义和使用，这里介绍如何使指向数组的指针变量作函数参数。

形式参数和实际参数均为指针变量。

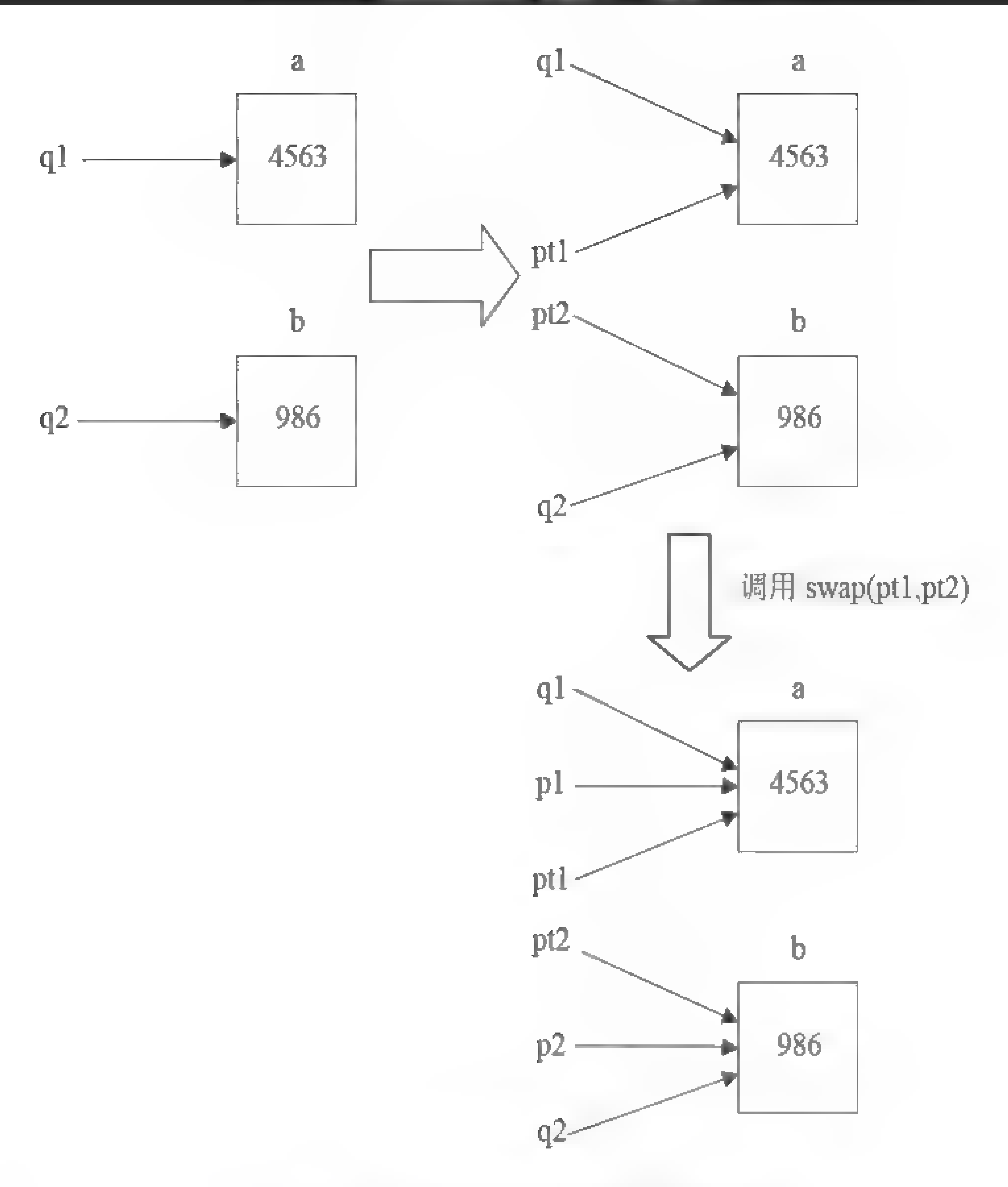


图 10.31　嵌套调用时指针的指向情况

【例 10.17】　任意输入 10 个数据，先将这 10 个数据中是奇数的数据输出，再求这 10 个数据中所有奇数之和。（**实例位置：资源包\TM\sl\10\17**）

```
#include<stdio.h>
void SUM(int *p,int n)                                  /*自定义函数 SUM 查找数组中的奇数*/
{
    int i,sum=0;
    printf("the odd:\n");
    for(i=0;i<n;i++)
        if(*(p+i)%2!=0)                                 /*判断数组中的元素是否为奇数*/
        {
            printf("%5d",*(p+i));
            sum=sum+*(p+i);
        }
        printf("\n");
        printf("sum:%d\n",sum);
}
main()
{
    int *pointer,a[10],i;
    pointer=a;                                          /*指针指向数组首地址*/
    printf("please input:\n");
```

```
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    SUM(pointer,10);                                    /*调用 SUM 函数*/
}
```

程序运行结果如图 10.32 所示。

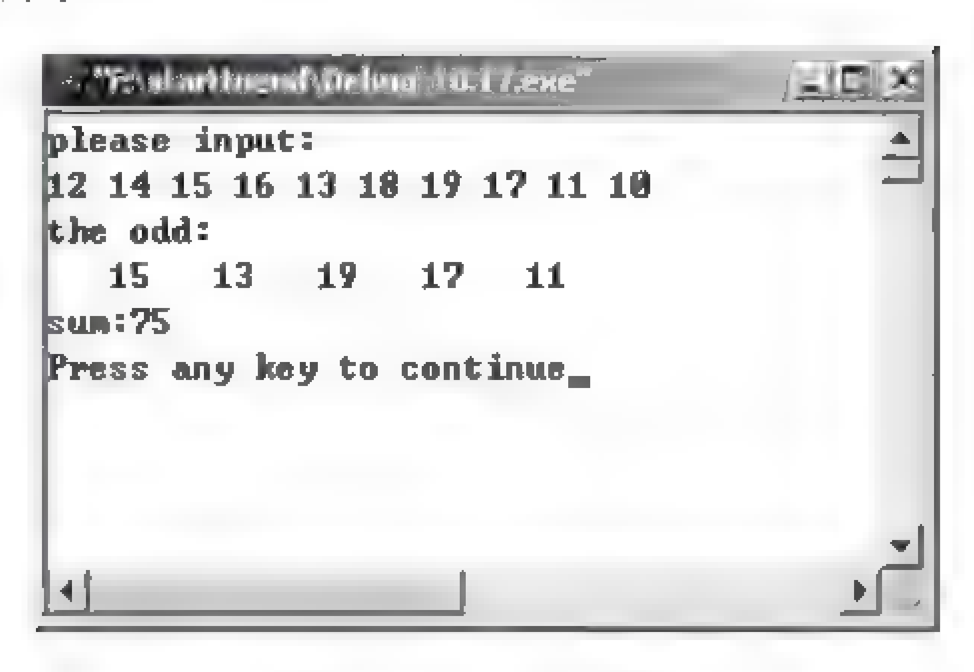


图 10.32　输出奇数

在自定义函数 SUM 中使用了指针变量作形式参数，在主函数中实际参数 pointer 是一个指向一维数组 a 的指针，虚实结合，被调用函数 SUM 中的形式参数 p 得到 pointer 的值，指向了内存中存放的一维数组。

冒泡排序是 C 语言中比较经典的例子，也是读者应该牢牢掌握的一种算法，下面分析如何使用指针变量作为函数参数来实现冒泡排序。

【例 10.18】　使用指针实现冒泡排序。（**实例位置：资源包\TM\sl\10\18**）

冒泡排序的基本思想：如果要对 n 个数进行冒泡排序，则要进行 n-1 轮比较，在第一轮比较中要进行 n-1 次两两比较，在第 j 轮比较中要进行 n-j 次两两比较。

```
#include<stdio.h>
void order(int *p,int n)
{
    int i,t,j;
    for(i=0;i<n-1;i++)
        for(j=0;j<n-1-i;j++)
            if(*(p+j)>*(p+j+1))                         /*判断相邻两个元素的大小*/
            {
                t=*(p+j);
                *(p+j)=*(p+j+1);
                *(p+j+1)=t;                             /*借助中间变量 t 进行值互换*/
            }
            printf("排序后的数组：");
            for(i=0;i<n;i++)
            {
                if(i%5==0)                              /*以每行 5 个元素的形式输出*/
                    printf("\n");
                printf("%5d",*(p+i));                   /*输出数组中排序后的元素*/
            }
            printf("\n");
```

```
}
main()
{
    int a[20],i,n;
    printf("请输入数组元素的个数：\n");
    scanf("%d",&n);                          /*输入数组元素的个数*/
    printf("请输入各个元素：\n");
    for(i=0;i<n;i++)
        scanf("%d",a+i);                     /*给数组元素赋初值*/
    order(a,n);                              /*调用 order 函数*/
}
```

程序运行结果如图 10.33 所示。

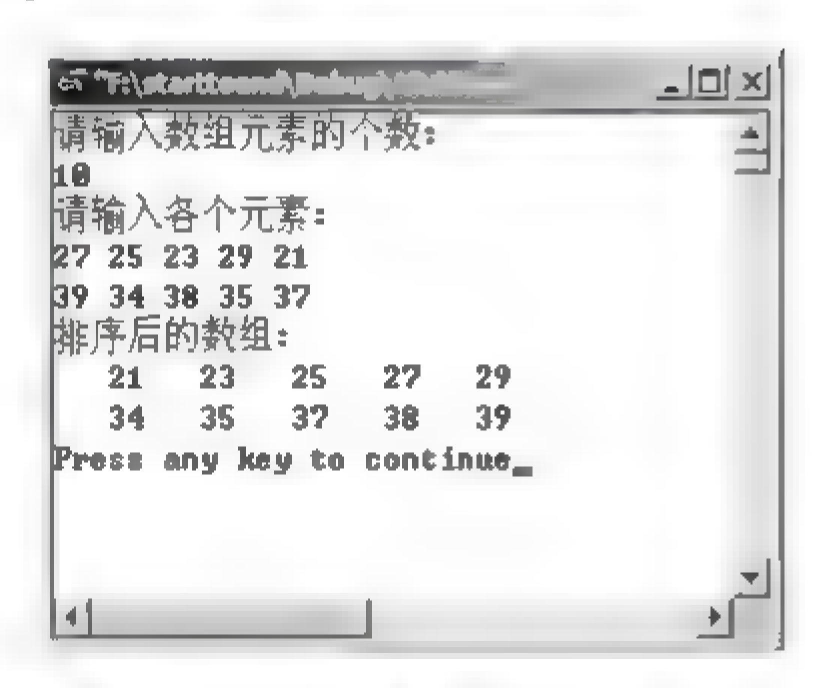


图 10.33　冒泡排序结果

前面两个实例都是用一个指向数组的指针变量作函数参数。在 10.3 节介绍过指向指针的指针，这里就来通过一个实例介绍如何用指向指针的指针作函数参数。

【例 10.19】　编程实现对英文的 12 个月份按字母顺序排序。（**实例位置：资源包\TM\sl\10\19**）

```
#include<stdio.h>
#include<string.h>
sort(char *strings[], int n)                 /*自定义排序函数*/
{
    char *temp;
    int i, j;
    for(i = 0; i < n; i++)
    {
        for(j = i + 1; j < n; j++)
        {
            if(strcmp(strings[i], strings[j]) > 0)    /*比较两个字符串的大小*/
            {
                temp = strings[i];
                strings[i] = strings[j];
                strings[j] = temp;           /*如果前面的字符比后面的大，则互换*/
            }
        }
    }
}
main()
{
```

```
    int n = 12;
    int i;
    char **p;                                   /*定义字符型指向指针的指针*/
    char *month[] =
    {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"

    };
    p = month;
    sort(p, n);                                 /*调用排序函数*/
    printf("排序后的 12 月份如下：\n");
    for (i = 0; i < n; i++)
        printf("%s\n", month[i]);               /*输出排序后的字符串*/
}
```

程序运行结果如图 10.34 所示。

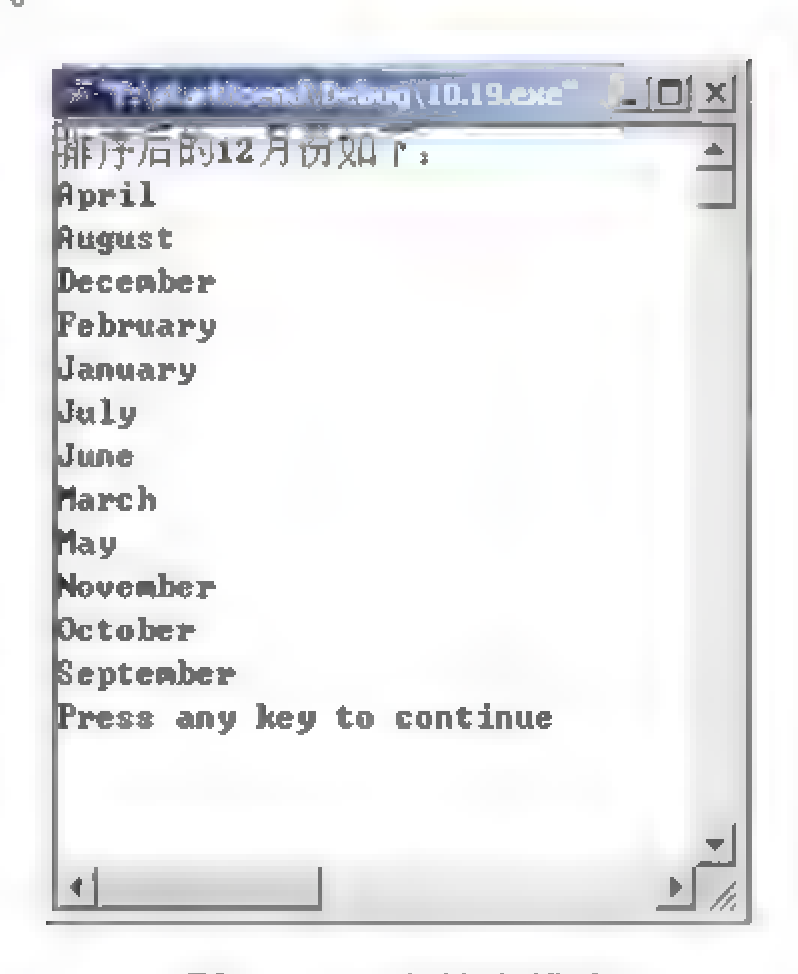


图 10.34　字符串排序

下面将通过一个二维数组使用指针变量作为函数参数的实例来加深读者对该部分知识的理解。

【例 10.20】　找出数组每行中最大的数，并将这些数相加求和。（**实例位置：资源包\TM\sl\10\20**）

```
#include<stdio.h>
#define N 4
void max(int (*a)[N],int m)                     /*自定义 max 函数，求数组每行的最大元素*/
{
```

```
    int value,i,j,sum=0;
    for(i=0;i<m;i++)
    {
        value=*(*(a+i));                          /*将一行中的首个元素赋给 value*/
        for(j=0;j<N;j++)
            if(*(*(a+i)+j)>value)                 /*判断其他元素是否小于 value*/
                value=*(*(a+i)+j);                /*把比 value 大的数重新赋给 value*/
            printf("第%d 行：最大数是：%d\n",i,value);
            sum=sum+value;
    }
    printf("\n");
    printf("每行中最大数相加之和是：%d\n",sum);
}
main()
{
    int a[3][N],i,j;
    int (*p)[N];
    p=&a[0];
    printf("please input:\n");
    for(i=0;i<3;i++)
        for(j=0;j<N;j++)
            scanf("%d",&a[i][j]);                 /*给数组中的元素赋值*/
        max(p,3);                                 /*调用 max 函数，指针变量作函数参数*/
}
```

程序运行结果如图 10.35 所示。



图 10.35　输出每行最大的数并求和

（1）数组名就是这个数组的首地址，因此也可以将数组名作为实参传递给形式参数。如例 10.18 中的语句：

```
order(a,n);                                       /*调用 order 函数*/
```

就是直接使用数组名作函数参数的。

（2）当形参为数组时，实参也可以为指针变量。可将例 10.18 改写成如下形式：

```
#include<stdio.h>
void order(int a[],int n)
{
```

```
	int i,t,j;
	for(i=0;i<n-1;i++)
		for(j=0;j<n-1-i;j++)
			if(*(a+j)>*(a+j+1))                 /*判断相邻两个元素的大小*/
			{
				t=*(a+j);
				*(a+j)=*(a+j+1);
				*(a+j+1)=t;                     /*借助中间变量 t 进行值互换*/
			}
			printf("排序后的数组：");
			for(i=0;i<n;i++)
			{
				if(i%5==0)                      /*以每行 5 个元素的形式输出*/
					printf("\n");
				printf("%5d",*(a+i));           /*输出数组中排序后的元素*/
			}
			printf("\n");
}
main()
{
	int a[20],i,n;
	int *p;
	p=a;
	printf("请输入数组元素的个数：\n");
	scanf("%d",&n);                             /*输入数组元素的个数*/
	printf("请输入各个元素：\n");
	for(i=0;i<n;i++)
		scanf("%d",p++);                        /*给数组元素赋初值*/
	p=a;
	order(p,n);                                 /*调用 order 函数*/
}
```

本程序中，形参是数组，而实参是指针变量。注意上述程序中倒数第 3 行语句：

```
p=a;
```

该语句不可少，如果将其省略，则后面调用 order 函数时，参数 p 指向的就不是 a 数组，这点需要读者加以注意。



10.5　返回指针值的函数

指针变量也可以指向一个函数。函数在编译时会被分配一个入口地址，该入口地址就称为函数的指针。可以用一个指针变量指向函数，然后通过该指针变量调用此函数。

一个函数可以带回一个整型值、字符值、实型值等，也可以带回指针型的数据，即地址。其概念与之前介绍的类似，只是带回的值的类型是指针类型而已。返回指针值的函数简称为指针函数。

定义指针函数的一般形式如下：

```
类型名 *函数名(参数表列);
```

例如：

```
int *fun(int x,int y)
```

fun 是函数名，调用它以后能得到一个指向整型数据的指针。x 和 y 是函数 fun 的形式参数，这两个参数均为基本整型。这个函数的函数名前面有一个“*”，表示此函数是指针型函数，类型说明是 int 表示返回的指针指向整型变量。

【例 10.21】　使用返回指针的函数查找最大值。（**实例位置：资源包\TM\sl\10\21**）

```
#include<stdio.h>
int per(int a,int b);
void main()
{
	int iWidth,iLength,iResult;
	printf("请输入长方形的长：\n");
	scanf("%d",&iLength);
	printf("请输入长方形的宽：\n");
	scanf("%d",&iWidth);
	iResult=per(iWidth,iLength);
	printf("长方形的周长是：");
	printf("%d\n",iResult);
}

int per(int a,int b)
{
	return (a+b)*2;
}
```

程序运行结果如图 10.36 所示。

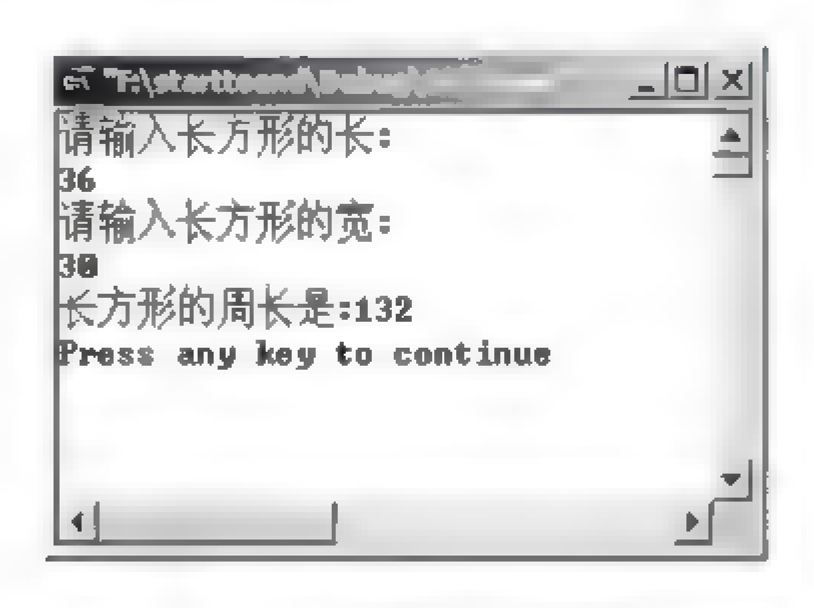


图 10.36　求长方形周长

例 10.21 中用前面讲过的方式自定义了一个 per 函数，用来求长方形的面积。下面就来看一下在例 10.21 的基础上如何使用返回值为指针的函数。

```
#include<stdio.h>
int *per(int a,int b);
```

```
int Perimeter;
void main()
{
	int iWidth,iLength;
	int *iResult;
	printf("请输入长方形的长：\n");
	scanf("%d",&iLength);
	printf("请输入长方形的宽：\n");
	scanf("%d",&iWidth);
	iResult=per(iWidth,iLength);
	printf("长方形的周长是：");
	printf("%d\n",*iResult);
}

int *per(int a,int b)
{
	int *p;
	p=&Perimeter;
	Perimeter=(a+b)*2;
	return p;
}
```

程序中自定义了一个返回指针值的函数：

```
int * per(int x,int y)
```

将指向存放着所求长方形周长的变量的指针变量返回。注意这个程序本身并不需要写成这种形式，因为对这种问题如上编写程序并不简便，这样写只是起到讲解的作用。



10.6 指针数组作 main 函数的参数

在前面讲过的程序中，几乎都会出现 main 函数。main 函数称为主函数，是运行所有程序的入口。main 函数是由系统调用的，当处于操作命令状态下，输入 main 所在的文件名，系统即调用 main 函数，在前面的内容中，对 main 函数始终作为主调函数进行处理，即允许 main 调用其他函数并传递参数。

main 函数的第 1 行一般形式如下：

```
main()
```

可以发现，main 函数是没有参数的。那么 main 函数能否有参数呢？实际上，main 函数可以是无参函数，也可以是有参的函数。对于有参的形式来说，就需要向其传递参数。下面先看一下 main 函数带参的形式：

```
main(int argc,char *argv[])
```

从函数参数的形式上看，包含一个整型和一个指针数组。当一个 C 的源程序经过编译、链接后，

会生成扩展名为.mp4 的可执行文件，这是可以在操作系统下直接运行的文件。对于 main 函数来说，其实际参数和命令是一起给出的，也就是一个命令行包括命令名和需要传给 main 函数的参数。命令行的一般形式如下：

```
命令名    参数 1    参数 2 … 参数 n
```

例如：

```
d:\debug\1 hello hi yeah
```

命令行中的命令就是可执行文件的文件名，如语句中的 d:\debug\1，命令名和其后所跟参数之间须用空格分隔。命令行与 main 函数的参数间存在一定关系。设命令行为：

```
file1 happy bright glad
```

其中，file1 为文件名，也就是一个由 file1.c 经编译、链接后生成的可执行文件 file1.mp4，其后各跟 3 个参数。以上命令行与 main 函数中的形式参数关系如下：

参数 argc 记录了命令行中命令与参数的个数（file1、happy、bright、glad），共 4 个，指针数组的大小由参数的值决定，即 char *argv[4]，该指针数组的取值情况如图 10.37 所示。

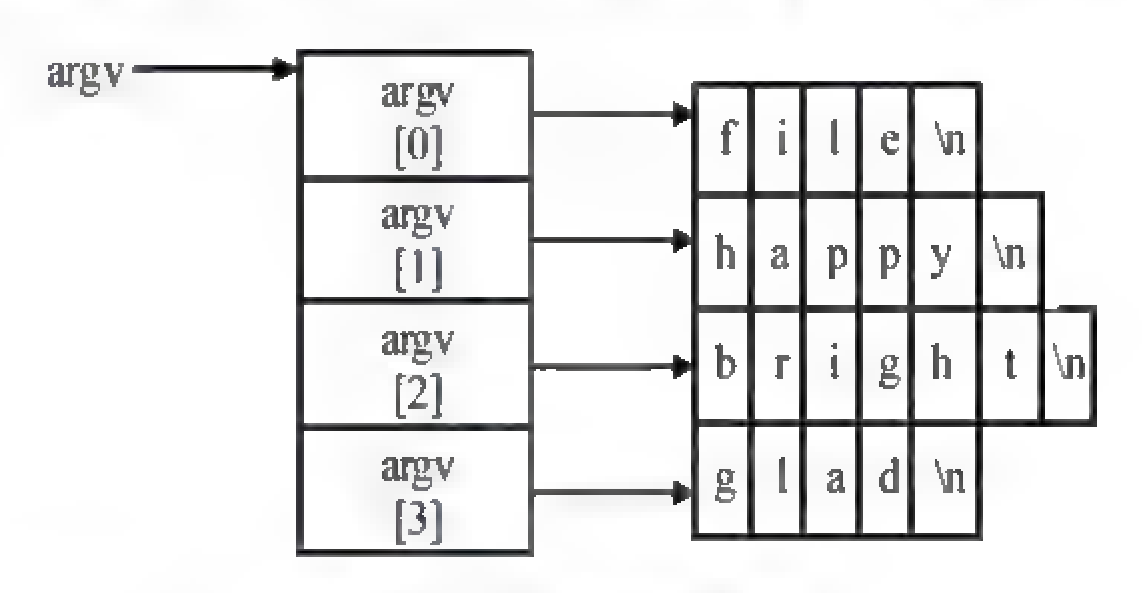


图 10.37　指针数组取值

利用指针数组作 main 函数的形参，可以向程序传送命令行参数。


说明

参数字符串的长度是不定的，并且参数字符串的长度不需要统一，且参数的数目也是任意的，并不规定具体个数。

下面通过例 10.22 具体介绍带参数的 main 函数是如何使用的。

【例 10.22】　输出 main 函数的参数内容。（**实例位置：资源包\TM\sl\10\22**）

```
#include<stdio.h>
main(int argc,char *argv[])                    /*main 函数为带参函数*/
{
    printf("the list of parameter:\n");
    printf("命令名：\n");
        printf("%s\n",*argv);
    printf("参数个数：\n");
    printf("%d\n",argc);

}
```

程序运行结果如图 10.38 所示。



图 10.38　输出参数内容

10.7　小　　结

本章主要介绍了指针的相关概念及其应用。首先要理解变量与指针之间的区别，重点掌握指针变量的相关概念及用法。指针与数组主要介绍了指针与一维数组、二维数组、字符串及字符串数组之间的关系，通常情况下把数组、字符串的首地址赋予指针变量。还讲解了指向指针的指针、如何使用指针变量作函数参数、返回指针值的函数，以及 main 函数的参数等相关内容，其中使用指针变量作为函数参数在编写程序过程中用得比较多，希望读者能够注意。

10.8　实践与练习

1．编程实现将数组中的元素值按照相反顺序存放。**（答案位置：资源包\TM\sl\10\23）**
2．输入两个字符串，将这两个字符串连接后输出。**（答案位置：资源包\TM\sl\10\24）**
3．使用指针实现字符串的复制，并将字符串输出。**（答案位置：资源包\TM\sl\10\25）**



高级应用

本篇介绍了结构体和共用体、位运算、预处理、文件、存储管理和网络套接字编程的内容。读者学习完这一部分，能够设计较复杂的程序，并且涉及的范围更广。

第11章

结构体和共用体

（视频讲解：40分钟）

迄今为止，我们在程序中用到的都是基本类型的数据。在编写程序时，简单的变量类型是不能满足程序中各种复杂数据的要求的，因此C语言还提供了构造类型的数据。构造类型数据是由基本类型按照一定规则组成的。

本章致力于使读者了解结构体的概念，掌握结构体和共用体的定义方式与使用方法，学会定义结构体数组、共用体数组、结构体指针及共用体指针，以及包含结构的结构。最后结合具体应用使大家对结构体和共用体有一个更为深刻的理解。

通过阅读本章，您可以：

- 了解结构体的概念
- 掌握定义结构体的方法
- 掌握结构体数组和结构体指针
- 了解链表的概念
- 熟悉链表的相关操作
- 掌握共用体
- 了解枚举类型



11.1 结　构　体

在此之前所介绍的类型都是基本类型，如整型 int、字符型 char 等，并且介绍了数组这种构造类型，数组中的各元素属于同一种类型。

但是在一些情况下，这些基本的类型是不能满足编写者的使用要求的。此时，程序员可以将一些有关的变量组织起来，定义成一个结构（structure），以此来表示一个有机的整体或一种新的类型。之后，程序就可以像处理内部的基本数据那样，对结构进行各种操作。

11.1.1　结构体类型的概念

结构体是一种构造类型，它是由若干成员组成的。其成员可以是一个基本数据类型，也可以是一个构造类型。既然结构体是一种新的类型，就需要先对其进行构造，这里称这种操作为声明一个结构体。声明结构体的过程就好比生产商品的过程，只有商品生产出来才可以使用该商品。

假如在程序中就要使用“商品”这样一个类型，一般的商品具有产品名称形状、颜色、功能、价格和产地等特点，如图 11.1 所示。

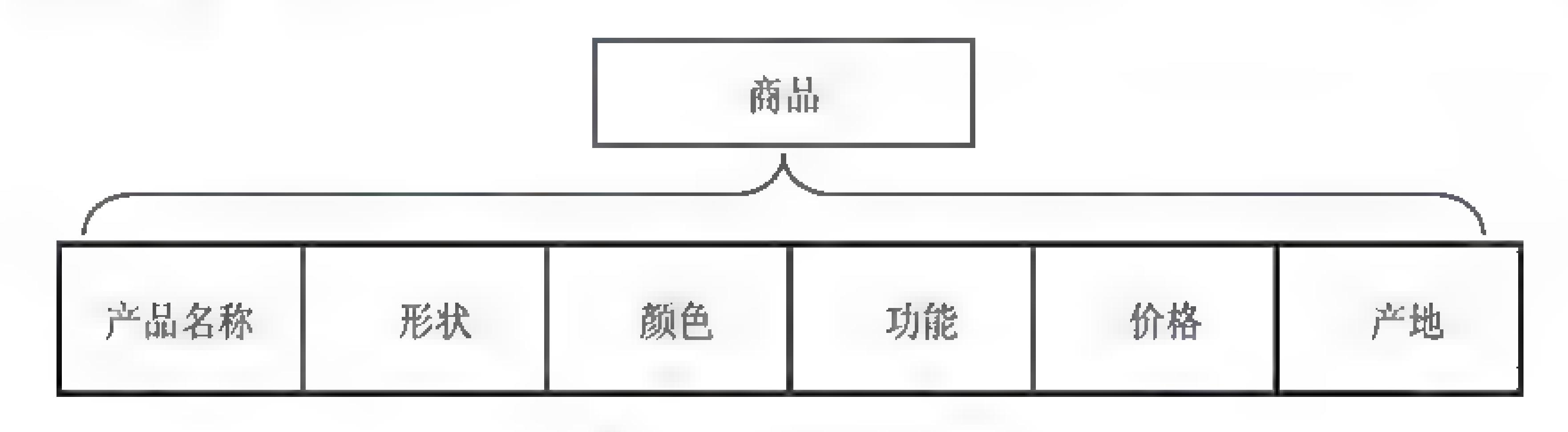


图 11.1　“商品”类型

通过图 11.1 可以看到，“商品”这种类型并不能使用之前学习过的任何一种类型表示，这时就要自己定义一种新的类型，将这种自己指定的结构称为结构体。

声明结构体时使用的关键字是 struct，其一般形式如下：



```
struct 结构体名
{
    成员列表
};
```

关键字 struct 表示声明结构，其后的结构体名表示该结构的类型名。大括号中的变量构成结构的成员，也就是一般形式中的成员列表处。

注意

在声明结构体时，要注意大括号最后面有一个分号“;”，在编程时千万不要忘记。

例如，声明一个结构体：

```
struct Product
{
	char cName[10];                          /*产品名称*/
	char cShape[20];                         /*形状*/
	char cColor[10];                         /*颜色*/
	char cFunc[20];                          /*功能*/
	int iPrice;                              /*价格*/
	char cArea[20];                          /*产地*/
};
```

上面的代码使用关键字 struct 声明一个名为 Product 的结构类型，在结构体中定义的变量是 Product 结构的成员，这些变量表示产品名称、形状、颜色、功能、价格和产地，可以根据结构成员中不同的作用选择与其相对应的类型。

11.1.2 结构体变量的定义

前面介绍了如何使用 struct 关键字来构造一个新的类型结构，以满足程序的设计要求。如何使用构造出来的类型才是构造新类型的目的。

声明一个结构体表示的是创建一种新的类型名，要用新的类型名再定义变量。定义的方式有 3 种：

（1）先声明结构体类型，再定义变量。

11.1.1 节中声明了一个 Product 结构体类型，接下来可以用 struct Product 定义两个结构体变量，例如：

```
struct Product product1;
struct Product product2;
```

struct Product 是结构体类型名，而 product1 和 product2 是结构体变量名。既然都是使用 Product 类型定义的变量，那么这两个变量就具有相同的结构。

定义一个基本类型的变量与定义一个结构体类型变量的不同之处在于：定义结构体变量不仅要求指定变量为结构体类型，而且要求指定为某一特定的结构体类型，如 struct Product；而定义基本类型的变量时（如整型变量），只需要指定 int 型即可。



说明

定义结构体变量后，系统就会为其分配内存单元。例如，product1 和 product2 在内存中各占 84 字节（10+20+10+20+4+20）。

技巧

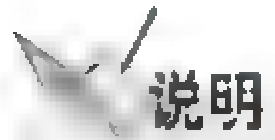
为了使规模较大的程序更便于修改和使用，常常将结构体类型的声明放在一个头文件中，这样在其他源文件中如果需要使用该结构体类型，则可以用#include 命令将该头文件包含到源文件中。

（2）在声明结构类型时，同时定义变量。

这种定义变量的一般形式如下：

```
struct 结构体名
{
    成员列表;
}变量名列表;
```

可以看到，在一般形式中将定义的变量的名称放在声明结构体的末尾处。但是需要注意的是，变量的名称要放在最后的分号前面。



说明

定义的变量不是只能有一个，可以定义多个变量。

例如，使用 struct Product 结构体类型名：

```
struct Product
{
    char cName[10];                                  /*产品名称*/
    char cShape[20];                                 /*形状*/
    char cColor[10];                                 /*颜色*/
    int iPrice;                                      /*价格*/
    char cArea[20];                                  /*产地*/
}product1,product2;                                  /*定义结构体变量*/
```

这种定义变量的方式与第一种方式相同，即定义了两个 struct Product 类型的变量 product1 和 product2。

（3）直接定义结构体类型变量。

其一般形式如下：

```
struct
{
成员列表
}变量名列表;
```

可以看出，这种方式没有给出结构体名称，如定义变量 product1 和 product2：

```
struct
{
    char cName[10];                                  /*产品名称*/
    char cShape[20];                                 /*形状*/
    char cColor[10];                                 /*颜色*/
    int iPrice;                                      /*价格*/
    char cArea[20];                                  /*产地*/
}product1,product2;                                  /*定义结构体变量*/
```

以上就是有关定义结构变量的 3 种方法。有关结构体的类型说明如下：

☑ 类型与变量是不同的。例如，只能对变量进行赋值操作，而不能对一个类型进行操作。这就像使用 int 型定义变量 iInt，可以为 iInt 进行赋值，但是不能为 int 进行赋值。在编译时，对类

型是不分配空间的，只对变量分配空间。

☑ 其中结构体的成员也可以是结构体类型的变量，例如：

```
struct date                          /*时间结构*/
{
    int year;                        /*年*/
    int month;                       /*月*/
    int day;                         /*日*/
};

struct student                       /*学生信息结构*/
{
    int num;                         /*学号*/
    char name[30];                   /*姓名*/
    char sex;                        /*性别*/
    int age;                         /*年龄*/
    struct date birthday;            /*出生日期*/
}student1,student2;
```

以上代码声明了一个时间的结构体类型，其中包括年、月、日；还声明了一个学生信息的结构类型，并且定义两个结构体变量 student1 和 student2。在 struct student 结构体类型中，可以看到有一个成员是表示学生的出生日期，使用的是 struct date 结构体类型。

11.1.3 结构体变量的引用

定义结构体类型变量以后，当然可以引用这个变量。但要注意的是，不能直接将一个结构体变量作为一个整体进行输入和输出。例如，不能将 product1 和 product2 进行以下输出：

```
printf("%s%s%s%d%s",product1);
printf("%s%s%s%d%s",product2);
```

要对结构体变量进行赋值、存取或运算，实质上就是对结构体成员进行操作。结构变量成员的一般形式如下：

```
结构变量名.成员名
```

在引用结构的成员时，可以在结构的变量名的后面加上成员运算符“.”和成员的名字。例如：

```
product1.cName="Icebox";
product2.iPrice=2000;
```

上面的赋值语句就是对 product1 结构体变量中的成员 cName 和 iPrice 两个变量进行赋值。

如果成员本身又属于一个结构体类型，应该怎么办呢？这时就要使用若干个成员运算符，一级一级地找到最低一级的成员。只能对最低级的成员进行赋值、存取以及运算操作。例如，对上面定义的 student1 变量中的出生日期进行赋值：

```
student1.birthday.year=1986;
student1.birthday.month=12;
student1.birthday.day=6;
```

注意

不能使用 student1.birthday 来访问 student1 变量中的成员 birthday，因为 birthday 本身也是一个结构体变量。

结构体变量的成员可以像普通变量一样，进行各种运算。例如：

```
product2.iPrice=product1.iPrice+500;
product1.iPrice++;
```

因为“.”运算符的优先级最高，所以 product1.iPrice++是 product1.iPrice 成员进行自加运算，而不是先对 iPrice 进行自加运算。

还可以对结构体变量成员的地址进行引用，也可以对结构体变量的地址进行引用，例如：

```
scanf("%d",&product1.iPrice);                /*输入成员 iPrice 的值*/
printf("%o",&product1);                      /*输出 product1 的首地址*/
```

【例 11.1】　引用结构体变量。(**实例位置：资源包\TM\sl\11\1**)

在本实例中声明结构体类型表示商品，然后定义结构体变量，之后对变量中的成员进行赋值，最后将结构体变量中保存的信息进行输出。

```
#include<stdio.h>

struct Product                               /*声明结构*/
{
    char cName[10];                          /*产品名称*/
    char cShape[20];                         /*形状*/
    char cColor[10];                         /*颜色*/
    int iPrice;                              /*价格*/
    char cArea[20];                          /*产地*/
};

int main()
{
    struct Product product1;                 /*定义结构体变量*/

    printf("please enter product's name\n"); /*信息提示*/
    scanf("%s",&product1.cName);             /*输出结构成员*/

    printf("please enter product's shape\n");/*信息提示*/
    scanf("%s",&product1.cShape);            /*输出结构成员*/

    printf("please enter product's color\n");/*信息提示*/
    scanf("%s",&product1.cColor);            /*输出结构成员*/
```

```
    printf("please enter product's price\n");        /*信息提示*/
    scanf("%d",&product1.iPrice);                    /*输出结构成员*/

    printf("please enter product's area\n");         /*信息提示*/
    scanf("%s",&product1.cArea);                     /*输出结构成员*/

    printf("Name: %s\n",product1.cName);             /*将成员变量输出*/
    printf("Shape: %s\n",product1.cShape);
    printf("Color: %s\n",product1.cColor);
    printf("Price: %d\n",product1.iPrice);
    printf("Area: %s\n",product1.cArea);

    return 0;
}
```

（1）在源文件中，先声明结构体变量类型用来表示商品这种特殊的类型，在结构体中定义了有关的成员。

（2）在主函数 main 中，使用 struct Product 定义结构体变量 product1，然后根据输出的信息提示，用户输入相应的结构成员数据。输入结构时成员在 scanf 函数中，引用了结构成员变量的地址&product1.cArea。

（3）当所有数据都输入完毕后，引用结构体变量 product1 中的成员，使用 printf 函数将其进行输出显示。

运行程序，显示效果如图 11.2 所示。

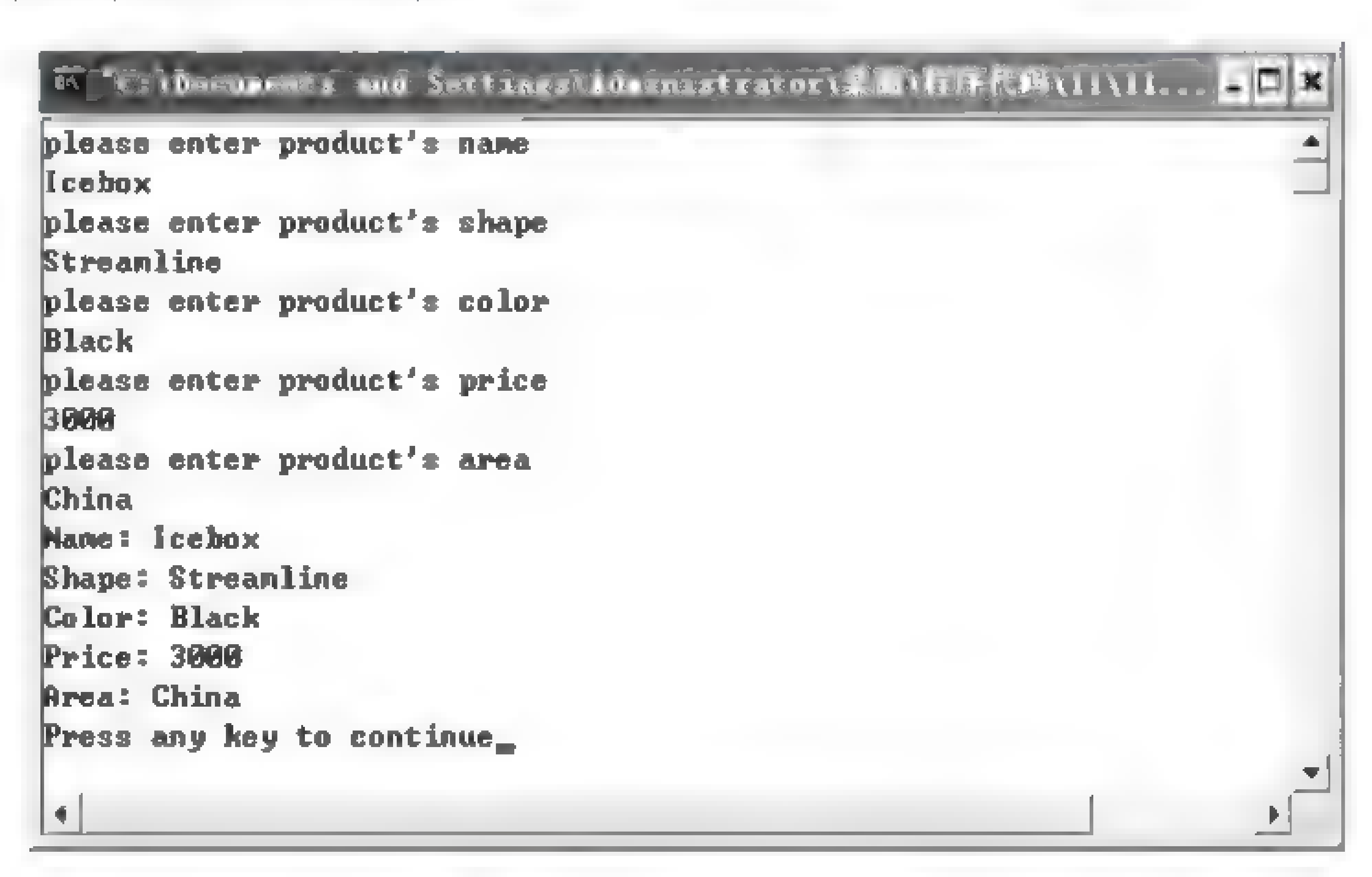


图 11.2　引用结构体变量

11.1.4　结构体类型的初始化

结构体类型与其他基本类型一样，也可以在定义结构体变量时指定初始值。例如：

```
struct Student
{
```

```
    char cName[20];
    char cSex;
    int iGrade;
} student1={"HanXue","W",3};                          /*定义变量并设置初始值*/
```

在初始化时要注意，定义的变量后面使用等号，然后将其初始化的值放在大括号中，并且每一个数据要与结构体的成员列表的顺序一样。

【例 11.2】　结构体类型的初始化操作。（**实例位置：资源包\TM\sl\11\2**）

本实例演示了两种初始化结构体的方式，一种是在声明结构及定义变量的同时进行初始化，另一种是在定义结构体变量后进行初始化。

```
#include<stdio.h>

struct Student                                        /*学生结构*/
{
    char cName[20];                                   /*姓名*/
    char cSex;                                        /*性别*/
    int iGrade;                                       /*年级*/
} student1={"HanXue",'W',3};                          /*定义变量并设置初始值*/


int main()
{
    struct Student student2={"WangJiasheng",'M',3};   /*定义变量并设置初始值*/

    /*将第一个结构体中的数据输出*/
    printf("the student1's information:\n");
    printf("Name: %s\n",student1.cName);
    printf("Sex: %c\n",student1.cSex);
    printf("Grade: %d\n",student1.iGrade);
    /*将第二个结构体中的数据输出*/
    printf("the student2's information:\n");
    printf("Name: %s\n",student2.cName);
    printf("Sex: %c\n",student2.cSex);
    printf("Grade: %d\n",student2.iGrade);
    return 0;
}
```

（1）从代码中可以看到，声明结构体时定义 student1 并且对其进行初始化操作，将要赋值的内容放在后面的大括号中，每一个数据都与结构体中的成员数据相对应。

（2）在 main 函数中，使用声明的结构体类型 struct Student 定义变量 student2，并且进行初始化的操作。

（3）最后将两个结构体变量中的成员进行输出，并比较二者数据的区别。

运行程序，显示效果如图 11.3 所示。

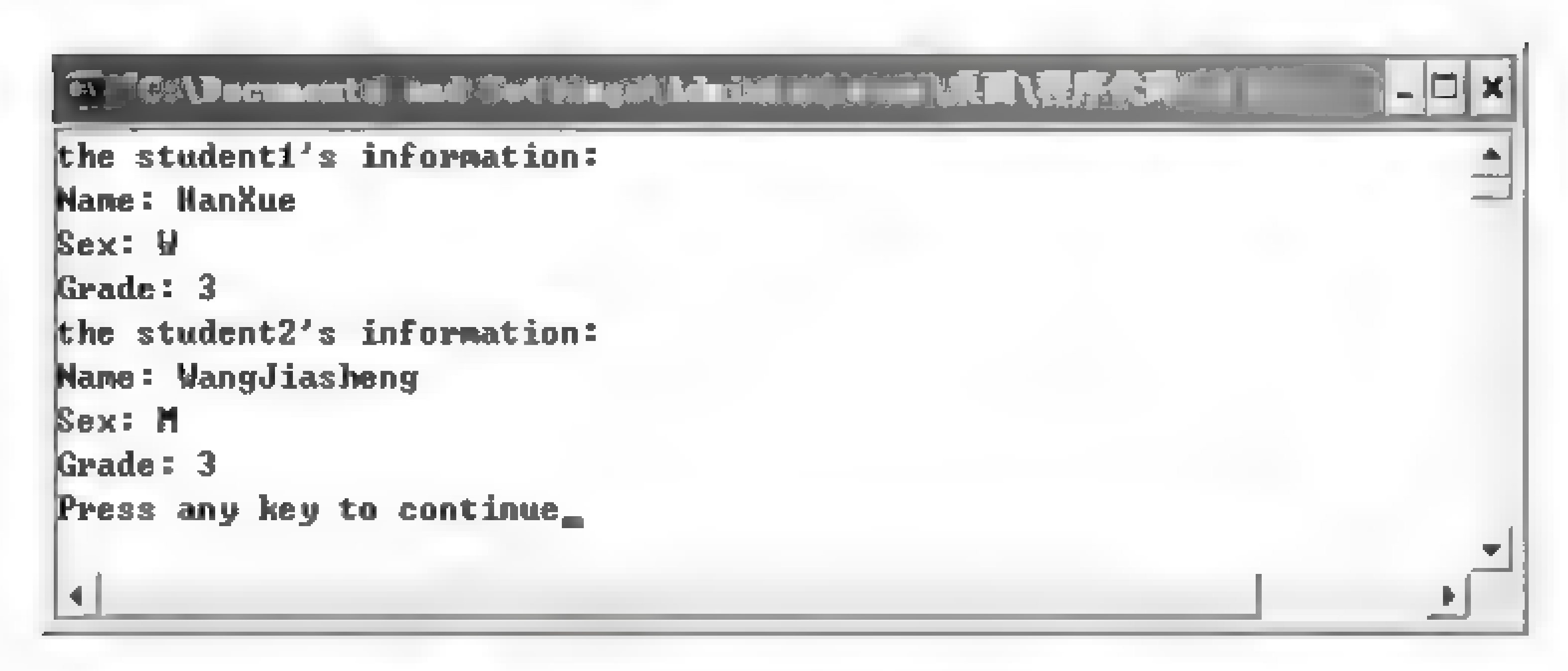

```
the student1's information:
Name: HanXue
Sex: W
Grade: 3
the student2's information:
Name: WangJiasheng
Sex: M
Grade: 3
Press any key to continue
```

图 11.3　结构体类型的初始化操作



11.2　结构体数组

当要定义 10 个整型变量时，前文介绍过可以将这 10 个变量定义成数组的形式。结构体变量中可以存放一组数据，例如一个学生信息包含姓名、性别和年级等。当需要定义 10 个学生的数据时，也可以使用数组的形式，这时称数组为结构体数组。

结构体数组与之前介绍的数组的区别就在于，数组中的元素是根据要求定义的结构体类型，而不是基本类型。

11.2.1　定义结构体数组

定义一个结构体数组的方式与定义结构体变量的方法相同，只是结构体变量替换成数组。定义结构体数组的一般形式如下：

```
struct 结构体名
{
    成员列表;
}数组名;
```

例如，定义学生信息的结构体数组，其中包含 5 个学生的信息：

```
struct Student                              /*学生结构*/
{
    char cName[20];                         /*姓名*/
    int iNumber;                            /*学号*/
    char cSex;                              /*性别*/
    int iGrade;                             /*年级*/
} student[5];                               /*定义结构体数组*/
```

这种定义结构体数组的方式是声明结构体类型的同时定义结构体数组，可以看到结构体数组和结构体变量的位置是相同的。

就像定义结构体变量那样，定义结构体数组也可以有不同的方式。例如，先声明结构体类型再定

义结构体数组：

```
struct Student student[5];                              /*定义结构体数组*/
```

或者直接定义结构体数组：

```
struct                                                  /*学生结构*/
{
    char cName[20];                                     /*姓名*/
    int iNumber;                                        /*学号*/
    char cSex;                                          /*性别*/
    int iGrade;                                         /*年级*/
} student[5];                                           /*定义结构体数组*/
```

上面的代码都是定义一个数组，其中的元素为 struct Student 类型的数据，每个数据中又有 4 个成员变量，如图 11.4 所示。

	cName	iNumber	cSex	iGrade
student[0]	WangJiasheng	12062212	M	3
student[1]	YuLongjiao	12062213	W	3
student[2]	JiangXuehuan	12062214	W	3
student[3]	ZhangMeng	12062215	W	3
student[4]	HanLiang	12062216	M	3

图 11.4　结构体数组

数组中各数据在内存中的存储是连续的，如图 11.5 所示。

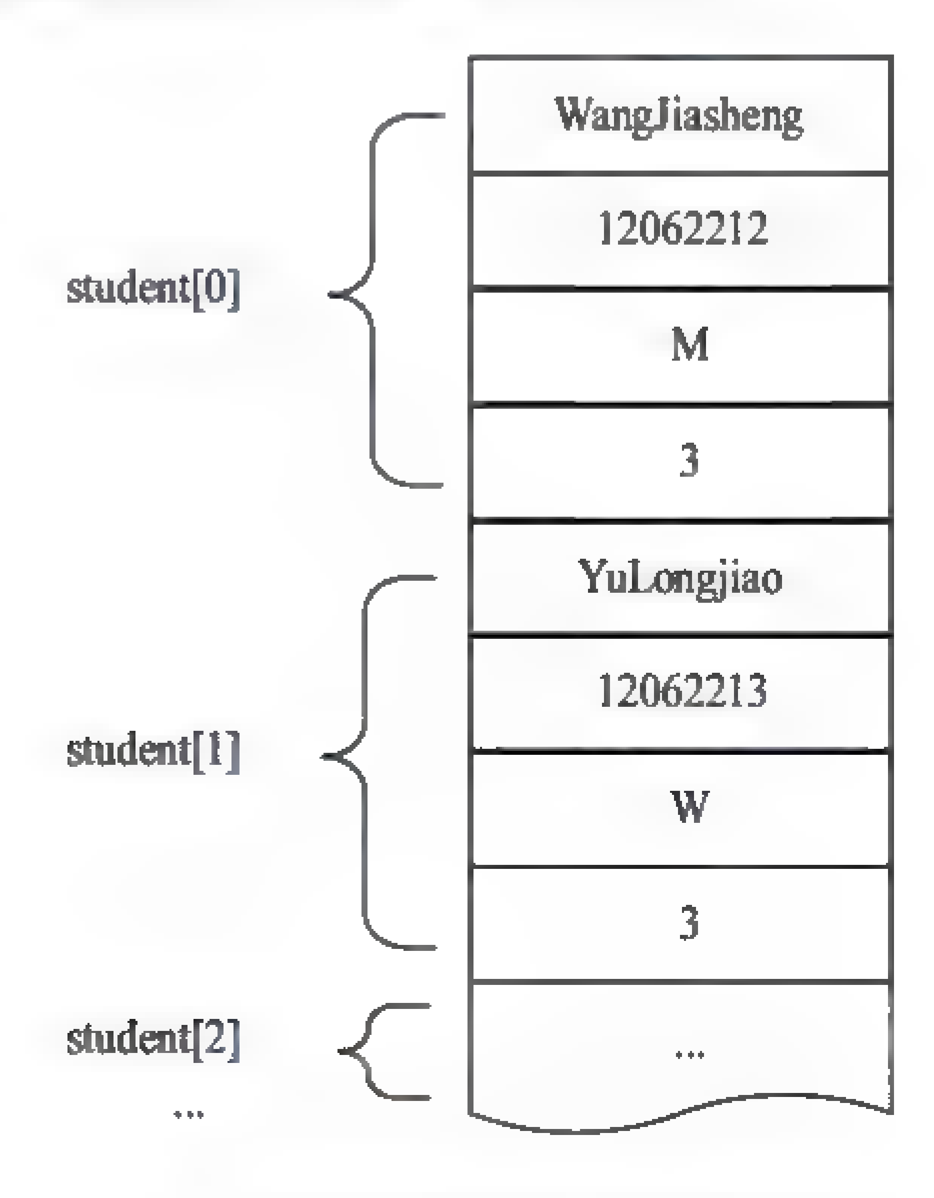


图 11.5　数组数据在内存中的存储形式

11.2.2 初始化结构体数组

与初始化基本类型的数组相同，也可以为结构体数组进行初始化操作。初始化结构体数组的一般形式如下：

```
struct 结构体名
{
    成员列表;
}数组名={初始值列表};
```

例如，为学生信息结构体数组进行初始化操作：

```
struct Student                              /*学生结构*/
{
    char cName[20];                         /*姓名*/
    int iNumber;                            /*学号*/
    char cSex;                              /*性别*/
    int iGrade;                             /*年级*/
} student[5]={{"WangJiasheng",12062212,'M',3},
            {"YuLongjiao",12062213,'W',3},
            {"JiangXuehuan",12062214,'W',3},
            {"ZhangMeng",12062215,'W',3},
            {"HanLiang",12062216,'M',3}};   /*定义数组并设置初始值*/
```

为数组进行初始化时，最外层的大括号表示所列出的是数组中的元素。因为每一个元素都是结构类型，所以每一个元素也使用大括号，其中包含每一个结构体元素的成员数据。

在定义数组 student 时，也可以不指定数组中的元素个数，这时编译器会根据数组后面的初始化值列表中给出的元素个数，来确定数组中元素的个数。例如：

```
student[ ]={...};
```

定义结构体数组时，可以先声明结构体类型，再定义结构体数组。同样，为结构体数组进行初始化操作时也可以使用同样的方式，例如：

```
struct student[5]={{"WangJiasheng",12062212,'M',3},
            {"YuLongjiao",12062213,'W',3},
            {"JiangXuehuan",12062214,'W',3},
            {"ZhangMeng",12062215,'W',3},
            {"HanLiang",12062216,'M',3}}
```

【例 11.3】 初始化结构体数组，并输出学生信息。（**实例位置：资源包\TM\sl\11\3**）

在本实例中，结构体数组通过初始化的方式保存学生信息。输出查看学生的信息，因为所查看的学生信息是一样的，因此可以使用循环操作。

```
#include<stdio.h>

struct Student                              /*学生结构*/
{
```

```
    char cName[20];                     /*姓名*/
    int iNumber;                        /*学号*/
    char cSex;                          /*性别*/
    int iGrade;                         /*年级*/
} student[5]={{"WangJiasheng",12062212,'M',3},
        {"YuLongjiao",12062213,'W',3},
        {"JiangXuehuan",12062214,'W',3},
        {"ZhangMeng",12062215,'W',3},
        {"HanLiang",12062216,'M',3}};   /*定义数组并设置初始值*/

int main()
{
    int i;                              /*循环控制变量*/
    for(i=0;i<5;i++)                    /*使用 for 进行 5 次循环*/
    {
        printf("NO%d student:\n",i+1);  /*首先输出学生的名次*/
        /*使用变量 i 作下标，输出数组中的元素数据*/
        printf("Name: %s, Number: %d\n",student[i].cName,student[i].iNumber);
        printf("Sex: %c, Grade: %d\n",student[i].cSex,student[i].iGrade);
        printf("\n");                   /*空格行*/
    }
    return 0;
}
```

（1）将学生所需要的信息声明为 struct Student 结构体类型，同时定义结构体数组 student，并为其初始化数据。需要注意的是，所给出数据的类型要与结构体中的成员变量的类型相符合。

（2）定义的数组包含 5 个元素，输出时使用 for 语句进行循环输出操作。其中，定义变量 i 为控制循环操作。因为数组的下标是从 0 开始的，所以为变量 i 赋值为 0。

（3）在 for 语句中，先显示每个学生的输出次序，其中因为 i 的初值为 0，所以要加上 1。之后将数组中的元素所表示的数据输出，这时变量 i 作为数组的下标，然后通过结构体成员的引用得到正确的数据，最后将其输出。

运行程序，显示效果如图 11.6 所示。

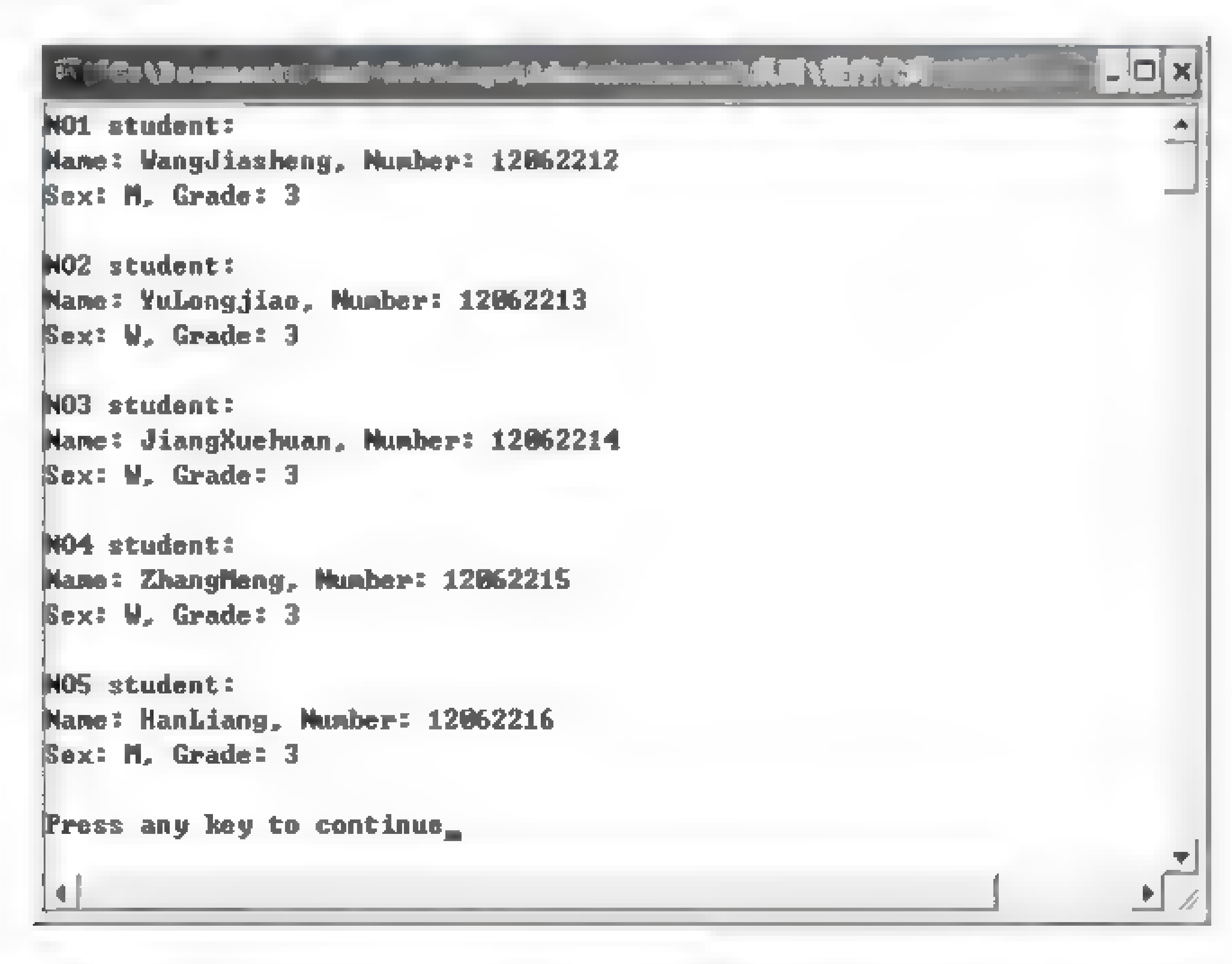


图 11.6 输出学生信息



11.3 结构体指针

一个指向变量的指针表示的是变量所占内存的起始地址。如果一个指针指向结构体变量，那么该指针指向的是结构体变量的起始地址。同样，指针变量也可以指向结构体数组中的元素。

11.3.1 指向结构体变量的指针

既然指针指向结构体变量的地址，因此可以使用指针来访问结构体中的成员。定义结构体指针的一般形式如下：

```
结构体类型 *指针名;
```

例如，定义一个指向 struct Student 结构类型的 pStruct 指针变量如下：

```
struct Student *pStruct;
```

使用指向结构体变量的指针访问成员有两种方法，pStruct 为指向结构体变量的指针。

第一种方法是使用点运算符引用结构成员：

```
(*pStruct).成员名
```

结构体变量可以使用点运算符对其中的成员进行引用。*pStruct 表示指向的结构体变量，因此使用点运算符可以应用结构体中的成员变量。

注意

*pStruct 一定要使用括号，因为点运算符的优先级是最高的，如果不使用括号，就会先执行点运算然后是*运算。

例如，pStruct 指针指向了 student1 结构体变量，引用其中的成员：

```
(*pStruct).iNumber=12061212;
```

【例 11.4】 通过指针使用点运算符引用结构体变量的成员。（**实例位置：资源包\TM\sl\11\4**）

本实例还使用之前声明过的学生结构。为结构体定义变量初始化赋值，然后使用指针指向该结构变量，最后通过指针引用变量中的成员进行显示。

```
#include<stdio.h>

int main()
{
    struct Student                          /*学生结构*/
    {
        char cName[20];                     /*姓名*/
```

```
        int iNumber;                                /*学号*/
        char cSex;                                  /*性别*/
        int iGrade;                                 /*年级*/
    }student={"SuYuQun",12061212,'W',2};            /*对结构变量进行初始化*/

    struct Student* pStruct;                        /*定义结构体类型指针*/
    pStruct=&student;                               /*指针指向结构体变量*/
    printf("----the student's information----\n");  /*消息提示*/
    printf("Name: %s\n",(*pStruct).cName);          /*使用指针引用变量中的成员*/
    printf("Number: %d\n",(*pStruct).iNumber);
    printf("Sex: %c\n",(*pStruct).cSex);
    printf("Grade: %d\n",(*pStruct).iGrade);
    return 0;
}
```

（1）首先在程序中声明结构类型，同时定义变量 student，为变量进行初始化的操作。

（2）定义结构体指针变量 pStruct，然后执行“pStruct=&student;”操作使得指针指向 student 变量。

（3）输出消息提示，然后在 printf 函数中使用指向结构变量的指针引用成员变量，将学生的信息进行输出。


说明

声明结构的位置可以放在 main 函数外也可以放在 main 函数内。

运行程序，显示效果如图 11.7 所示。



图 11.7　通过指针使用点运算符引用结构体变量的成员

第二种方法是使用指向运算符引用结构成员：

```
pStruct ->成员名;
```

例如，使用指向运算符引用一个变量的成员：

```
pStruct->iNumber=12061212;
```

假如 student 为结构体变量，pStruct 为指向结构体变量的指针，可以看出以下 3 种形式的效果是等价的。

☑ student.成员名。

☑ (*pStruct).成员名。

☑ pStruct->成员名。

注意

在使用“->”引用成员时，要注意分析以下情况：

（1）pStruct->iGrade，表示指向的结构体变量中成员 iGrade 的值。

（2）pStruct->iGrade++，表示指向的结构体变量中成员 iGrade 的值，使用后该值加 1。

（3）++pStruct->iGrade，表示指向的结构体变量中成员 iGrade 的值加 1，计算后再进行使用。

【例 11.5】 使用指向运算符引用结构体对象成员。**（实例位置：资源包\TM\sl\11\5）**

在本实例中，定义结构体变量但不对其进行初始化操作，使用指针指向结构体变量并为其成员进行赋值操作。

```
#include<stdio.h>
#include<string.h>

struct Student                                  /*学生结构*/
{
    char cName[20];                             /*姓名*/
    int iNumber;                                /*学号*/
    char cSex;                                  /*性别*/
    int iGrade;                                 /*年级*/
}student;                                       /*定义变量*/

int main()
{
    struct Student* pStruct;                    /*定义结构体类型指针*/
    pStruct=&student;                           /*指针指向结构体变量*/

    strcpy(pStruct->cName,"SuYuQun");           /*将字符串常量复制到成员变量中*/
    pStruct->iNumber=12061212;                  /*为成员变量赋值*/
    pStruct->cSex='W';
    pStruct->iGrade=2;

    printf("-----the student's information-----\n");  /*消息提示*/
    printf("Name: %s\n",student.cName);         /*使用变量直接输出*/
    printf("Number: %d\n",student.iNumber);
    printf("Sex: %c\n",student.cSex);
    printf("Grade: %d\n",student.iGrade);
    return 0;
}
```

（1）在程序中使用了 strcpy 函数将一个字符串常量复制到成员变量中，使用该函数要在程序中包含头文件 string.h。

（2）可以看到在为成员赋值时，使用的是指向运算符引用的成员变量，在程序的最后使用结构体变量和点运算符直接将成员的数据进行输出。输出的结果表示使用指向运算符为成员变量赋值成功。

运行程序，显示效果如图 11.8 所示。

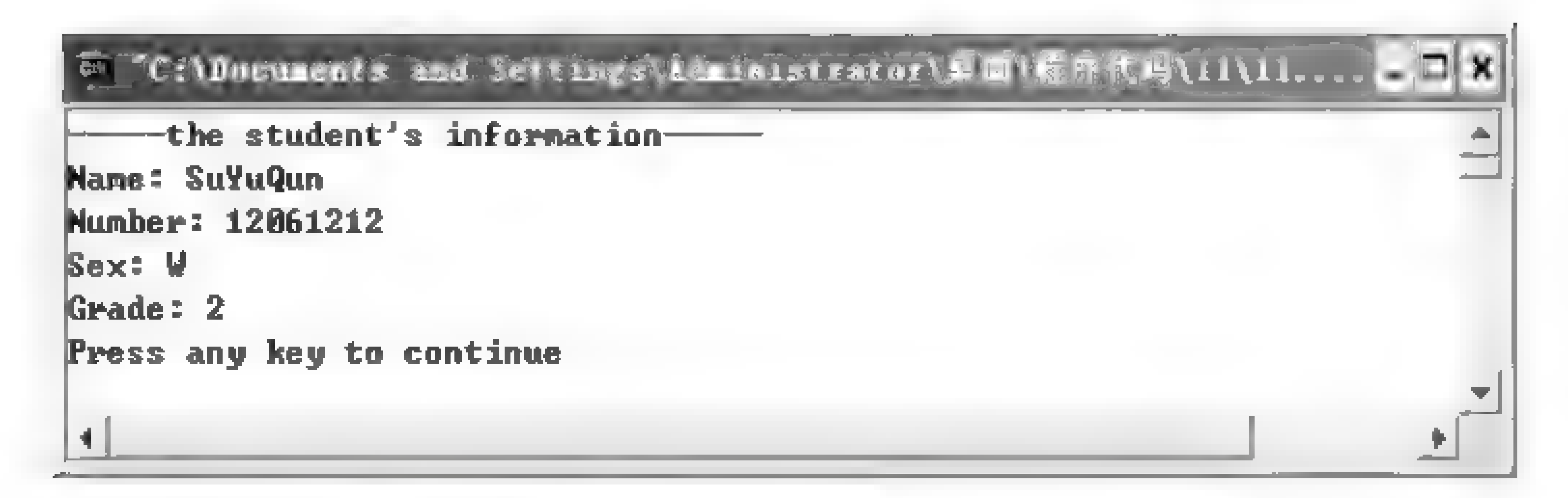


图 11.8　使用指向运算符引用结构体对象成员

11.3.2　指向结构体数组的指针

结构体指针变量不但可以指向一个结构体变量，还可以指向结构体数组，此时指针变量的值就是结构体数组的首地址。

结构体指针变量也可以直接指向结构体数组中的元素，这时指针变量的值就是该结构体数组元素的首地址。例如，定义一个结构体数组 student[5]，使用结构体指针指向该数组：

```
struct Student* pStruct;
pStruct=student;
```

因为数组不使用下标时表示的是数组的第一个元素的地址，所以指针指向数组的首地址。如果想利用指针指向第 3 个元素，则在数组名后附加下标，然后在数组名前使用取地址符号&，例如：

```
pStruct=&student[2];
```

【例 11.6】　使用结构体指针变量指向结构体数组。（**实例位置：资源包\TM\sl\11\6**）

在本实例中，使用之前声明的学生结构类型定义结构体数组，并对其进行初始化操作。通过指向该数组的指针，将其中元素的数据进行输出显示。

```
#include<stdio.h>

struct Student                                  /*学生结构*/
{
    char cName[20];                             /*姓名*/
    int iNumber;                                /*学号*/
    char cSex;                                  /*性别*/
    int iGrade;                                 /*年级*/
} student[5]={{"WangJiasheng",12062212,'M',3},
        {"YuLongjiao",12062213,'W',3},
        {"JiangXuehuan",12062214,'W',3},
        {"ZhangMeng",12062215,'W',3},
        {"HanLiang",12062216,'M',3}};           /*定义数组并设置初始值*/

int main()
{
    struct Student* pStruct;
    int index;
```

```
    pStruct=student;
    for(index=0;index<5;index++,pStruct++)
    {
        printf("NO%d student:\n",index+1);          /*首先输出学生的名次*/
        /*使用变量 index 做下标，输出数组中的元素数据*/
        printf("Name: %s, Number: %d\n",pStruct->cName,pStruct->iNumber);
        printf("Sex: %c, Grade: %d\n",pStruct->cSex,pStruct->iGrade);
        printf("\n");                                /*空格行*/
    }
    return 0;
}
```

（1）在代码中定义了一个结构体数组 student[5]，定义结构体指针变量 pStruct 指向该数组的首地址。

（2）使用 for 语句，对数组元素进行循环操作。在循环语句块中，pStruct 刚开始是指向数组的首地址，也就是第一个元素的地址，因此使用 pStruct->引用的是第一个元素中的成员。使用输出函数显示成员变量表示的数据。

（3）当一次循环语句结束之后，循环变量进行自加操作，同时 pStruct 也执行自加运算。这里需要注意的是，pStruct++表示 pStruct 的增加值为一个数组元素的大小，也就是说 pStruct++表示的是数组元素中的第二个元素 student[1]。

注意

(++pStruct)->Number 与(pStruct++)->Number 的区别在于，前者是先执行++操作，使得 pStruct 指向下一个元素的地址，然后取得该元素的成员值；而后者是先取得当前元素的成员值，再使得 pStruct 指向下一个元素的地址。

运行程序，显示效果如图 11.9 所示。

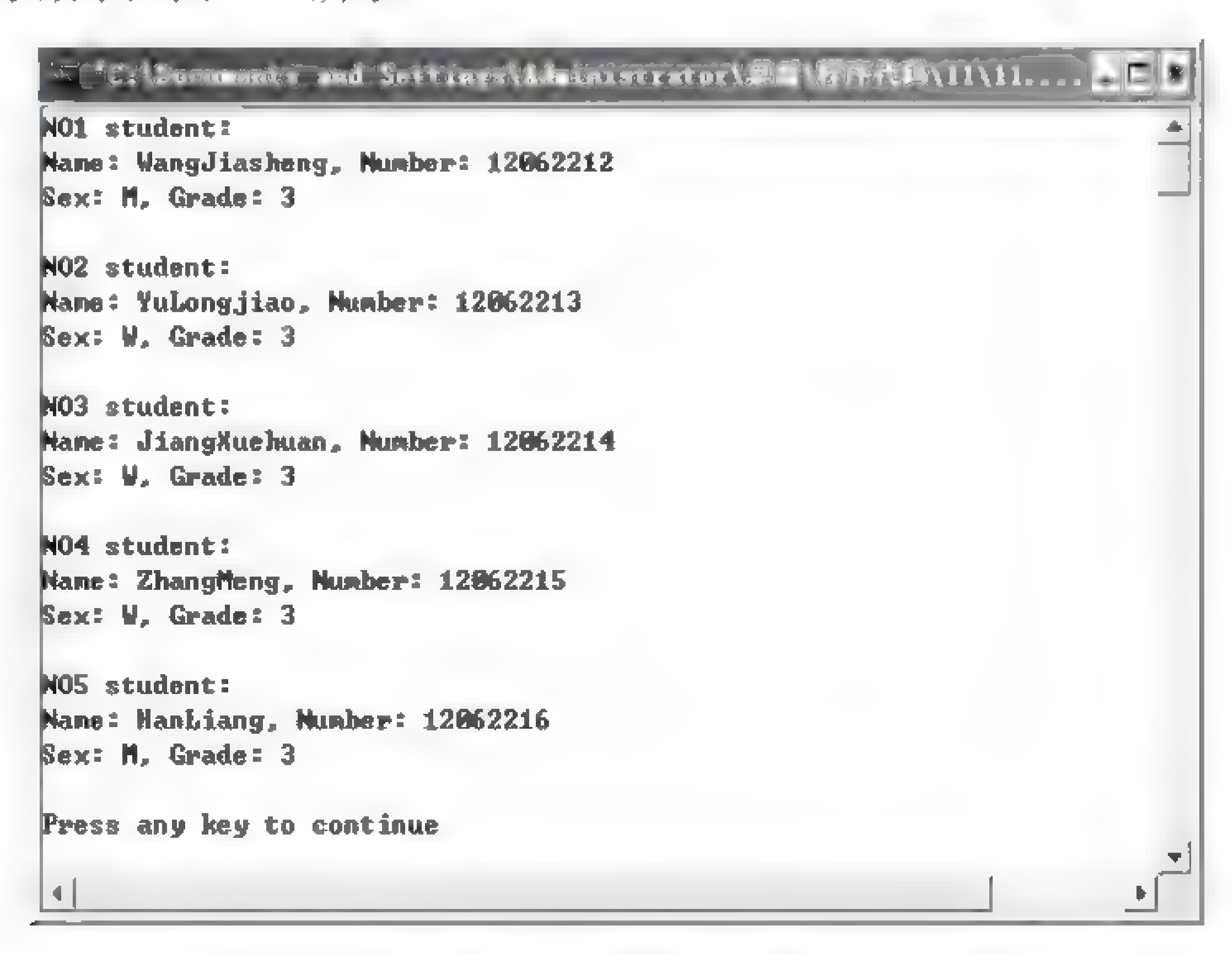


图 11.9　使用结构体指针变量指向结构体数组

11.3.3　结构体作为函数参数

函数是有参数的，可以将结构体变量的值作为一个函数的参数。使用结构体作为函数的参数有 3 种形式：使用结构体变量作为函数参数；使用指向结构体变量的指针作为函数参数；使用结构体变量的成员作为函数参数。

1．使用结构体变量作为函数参数

使用结构体变量作为函数的实参时，采取的是“值传递”方式，即会将结构体变量所占内存单元的内容全部顺序传递给形参，形参也必须是同类型的结构体变量。例如：

```
void Display(struct Student stu);
```

在形参的位置使用结构体变量，但是函数调用期间，形参也要占用内存单元。这种传递方式在空间和时间上开销都比较大。

另外，根据函数参数传值方式，如果在函数内部修改了变量中成员的值，则改变的值不会返回到主调函数中。

【例 11.7】　使用结构体变量作为函数参数。（**实例位置：资源包\TM\sl\11\7**）

在本实例中，声明一个简单的结构类型表示学生成绩，编写一个函数，使得该结构类型变量作为函数的参数。

```
#include<stdio.h>

struct Student                                    /*学生结构*/
{
    char cName[20];                               /*姓名*/
    float fScore[3];                              /*分数*/
}student={"SuYuQun",98.5f,89.0,93.5f};            /*定义变量*/

void Display(struct Student stu)                  /*形参为结构体变量*/
{
    printf("----Information----\n");             /*提示信息*/
    printf("Name: %s\n",stu.cName);               /*引用结构成员*/
    printf("Chinese: %.2f\n",stu.fScore[0]);
    printf("Math: %.2f\n",stu.fScore[1]);
    printf("English: %.2f\n",stu.fScore[2]);
    /*计算平均分数*/
    printf("Average score:%.2f\n",(stu.fScore[0]+stu.fScore[1]+stu.fScore[2])/3);
}

int main()
{
    Display(student);                             /*调用函数，结构变量作为实参进行传递*/
    return 0;
}
```

（1）在程序中声明一个简单的结构体表示学生的分数信息，在这个结构体中定义一个字符数组表示名称，还定义了一个实型数组表示 3 个学科的分数。在声明结构的最后同时定义变量，并进行初始化。

（2）之后定义一个名为 Display 的函数，其中用结构体变量作为函数的形式参数。在函数体中，使用参数 stu 引用结构中的成员，输出学生的姓名和 3 个学科的成绩，并在最后通过表达式计算出平均成绩。

（3）在主函数 main 中，使用 student 结构体变量作为参数，调用 Display 函数。

运行程序，显示效果如图 11.10 所示。



图 11.10　使用结构体变量作为函数参数

2. 使用指向结构体变量的指针作为函数参数

在使用结构体变量作为函数的参数时，在传值的过程中空间和时间的开销比较大，那么有没有一种更好的传递方式呢？有！就是使用结构体变量的指针作为函数的参数进行传递。

在传递结构体变量的指针时，只是将结构体变量的首地址进行传递，并没有将变量的副本进行传递。例如，声明一个传递结构体变量指针的函数如下：

```
void Display(struct Student* stu)
```

这样使用形参 stu 指针就可以引用结构体变量中的成员了。这里需要注意的是，因为传递的是变量的地址，如果在函数中改变成员中的数据，那么返回主调函数时变量会发生改变。

【例 11.8】　使用结构体变量指针作为函数参数。（**实例位置：资源包\TM\sl\11\8**）

本实例对例 11.7 做了一点小的改动，其中使用结构体变量的指针作为函数的参数，并且在函数中改动结构体成员的数据。通过前后两次的输出，比较二者的区别。

```
#include<stdio.h>

struct Student                                          /*学生结构*/
{
    char cName[20];                                     /*姓名*/
    float fScore[3];                                    /*分数*/
}student={"SuYuQun",98.5f,89.0,93.5f};                  /*定义变量*/

void Display(struct Student* stu)                       /*形参为结构体变量的指针*/
{
    printf("——Information——\n");                        /*提示信息*/
    printf("Name: %s\n",stu->cName);                    /*使用指针引用结构体变量中的成员*/
    printf("English: %.2f\n",stu->fScore[2]);           /*输出英语的分数*/
```

```
    stu->fScore[2]=90.0f;                               /*更改成员变量的值*/
}

int main()
{
    struct Student* pStruct=&student;                   /*定义结构体变量指针*/
    Display(pStruct);                                   /*调用函数，结构变量作为实参进行传递*/
    printf("Changed English: %.2f\n",pStruct->fScore[2]);  /*输出成员的值*/
    return 0;
}
```

（1）在本实例中，函数的参数是结构体变量的指针，因此在函数体中要通过使用指向运算符“->”引用成员的数据。为了简化操作，只将英语成绩进行输出，并且最后更改成员的数据。

（2）在主函数 main 中，先定义结构体变量指针，并将结构体变量的地址传递给指针，将指针作为函数的参数进行传递。函数调用完后，再显示一次变量中的成员数据。通过输出结果可以看到，在函数中通过指针改变成员的值，在返回主调用函数中值发生变化。



说明

在程序中，为了直观地看出函数传递的参数是结构体变量的指针，定义了一个指针变量指向结构体。实际上，可以直接传递结构体变量的地址作为函数的参数，如“Display(&student);”。

运行程序，显示效果如图 11.11 所示。

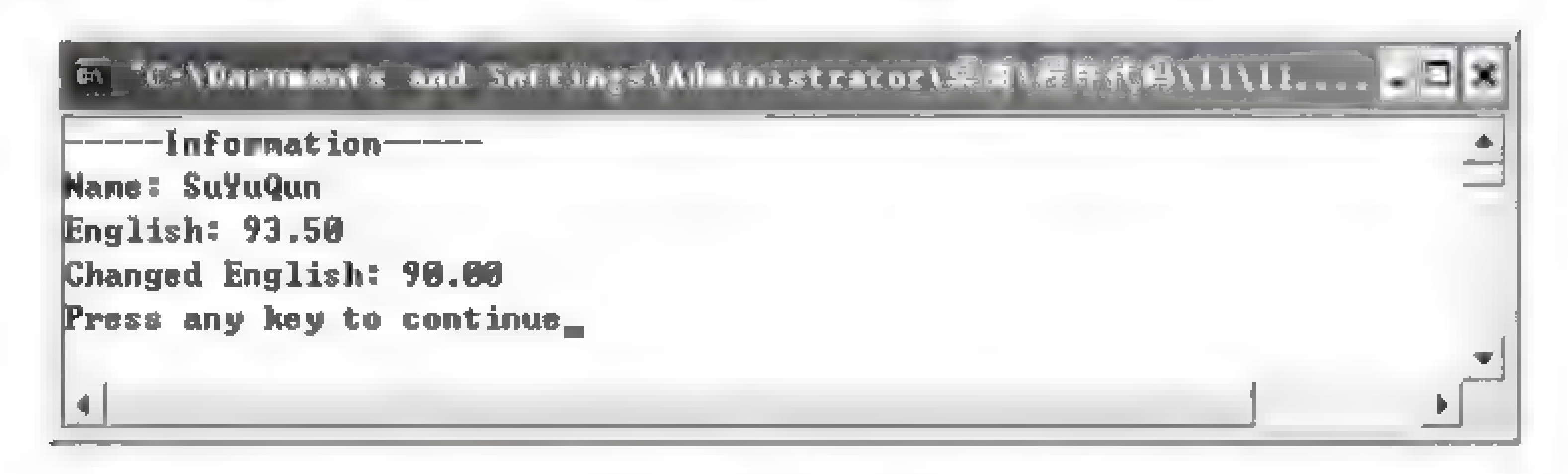


图 11.11　使用结构体变量指针作为函数参数

3. 使用结构体变量的成员作为函数参数

使用这种方式为函数传递参数与普通的变量作为实参是一样的，是传值方式传递。例如：

```
Display(student.fScore[0]);
```



注意

传值时，实参要与形参的类型一致。



11.4　包含结构的结构

在介绍有关结构体变量的定义时曾经说过，结构体中的成员不仅可以是基本类型，也可以是结构

体类型。

例如，定义一个学生信息结构体类型，其中的成员包括姓名、学号、性别、出生日期。其中，成员出生日期又属于一个结构体类型，因为出生日期包括年、月、日这 3 个成员。这样，学生信息这个结构体类型就是包含结构的结构。

【例 11.9】 包含结构的结构。（**实例位置：资源包\TM\sl\11\9**）

在本实例中，定义两个结构体类型，一个表示日期，一个表示学生的个人信息。其中，日期结构体是个人信息结构中的成员。通过使用个人信息结构类型表示学生的基本信息内容。

```
#include<stdio.h>

struct date                                             /*时间结构*/
{
    int year;                                           /*年*/
    int month;                                          /*月*/
    int day;                                            /*日*/
};

struct student                                          /*学生信息结构*/
{
    char name[30];                                      /*姓名*/
    int num;                                            /*学号*/
    char sex;                                           /*性别*/
    struct date birthday;                               /*出生日期*/
}student={"SuYuQun",12061212,'W',{1986,12,6}};          /*为结构变量初始化*/

int main()
{
    printf("----Information----\n");
    printf("Name: %s\n",student.name);                  /*输出结构成员*/
    printf("Number: %d\n",student.num);
    printf("Sex: %c\n",student.sex);
    printf("Birthday: %d,%d,%d\n",student.birthday.year,
            student.birthday.month,student.birthday.day); /*将成员结构体数据输出*/
    return 0;
}
```

（1）程序中在为包含结构的结构 struct student 类型初始化时要注意，因为出生日期是结构体，所以要使用大括号将赋值的数据包含在内。

（2）在引用成员结构体变量的成员时，例如，student.birthday.year、student.birthday 表示引用 student 变量中的成员 birthday，因此 student.birthday.year 表示 student 变量中结构体变量 birthday 的成员 year 变量的值。

运行程序，显示效果如图 11.12 所示。



图 11.12　包含结构的结构



11.5 链　　表

数据是信息的载体，是描述客观事物属性的数、字符，以及所有能输入计算机中并被计算机程序识别和处理的集合。数据结构是指数据对象以及其中的相互关系和构造方法。在数据结构中有一种线性存储结构称为线性表，本节将根据前面所学的结构体的知识介绍有关线性表的链式存储结构，也称其为链表。

11.5.1　链表概述

链表是一种常见的数据结构。前面介绍过使用数组存放数据，但是使用数组时要先指定数组中包含元素的个数，即为数组的长度。但是如果向这个数组中加入的元素个数超过了数组的大小时，便不能将内容完全保存。例如，在定义一个班级的人数时，如果小班是 30 人，普通班级是 50 人，且定义班级人数时使用的是数组，那么要定义数组的个数为最大，也就是最少为 50 个元素，否则不满足最大时的情况。这种方式非常浪费空间。

这时就希望有一种存储方式，其存储元素的个数是不受限定的，当进行添加元素时，存储的个数就会随之改变，这种存储方式就是链表。

如图 11.13 所示为链表结构的示意图。

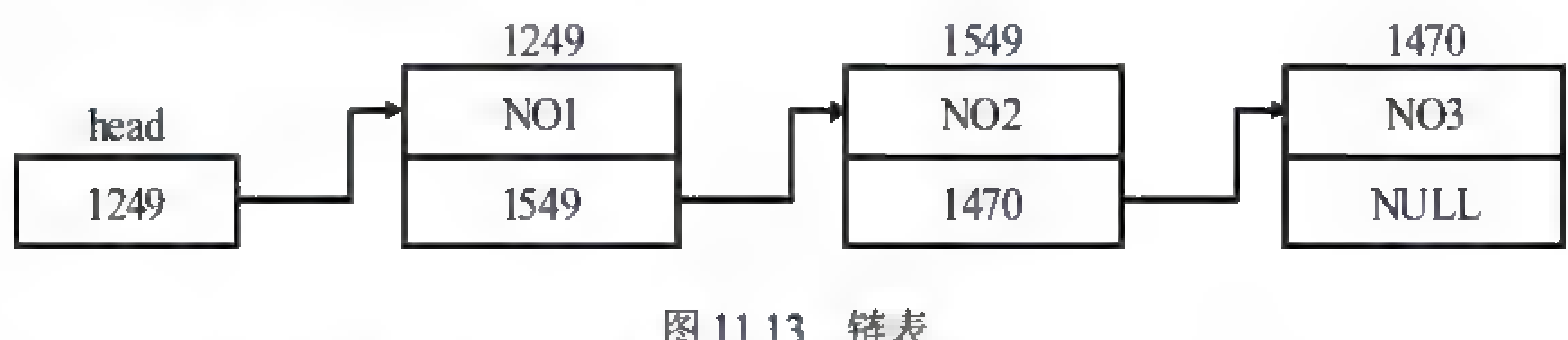


图 11.13　链表

在链表中有一个头指针变量，图 11.13 中 head 表示的就是头指针，这个指针变量保存一个地址。从图 11.13 中的箭头可以看到，该地址为一个变量的地址，也就是说头指针指向一个变量，这个变量称为元素。在链表中每一个元素包括数据部分和指针部分。数据部分用来存放元素所包含的数据，而指针部分用来指向下一个元素。最后一个元素的指针指向 NULL，表示指向的地址为空。

从链表的示意图可以看到，head 头节点指向第一个元素，第一个元素中的指针又指向第二个元素，第二个元素的指针又指向第 3 个元素的地址，第 3 个元素的指针就指向为空。

根据对链表的描述，可以想象链表就像一个铁链，一环扣一环，然后通过头指针寻找链表中的元素。这就好比在一个幼儿园中，老师拉着第一个小朋友的手，第一个小朋友又拉着第二个小朋友的手，这样下去，幼儿园中的小朋友就连成了一条线。最后一个小朋友没有拉着任何人，他的手是空着的，他就好像是链表中的链尾，而老师就是头指针，通过老师就可以找到这个队伍中的任何一个小朋友。

注意

在链表这种数据结构中，必须利用指针才能实现，因此链表中的节点应该包含一个指针变量来保存下一个节点的地址。

例如，设计一个链表表示一个班级，链表中的节点表示学生：

```
struct Student
{
	char cName[20];			/*姓名*/
	int iNumber;			/*学号*/
	struct Student* pNext;		/*指向下一个节点的指针*/
};
```

可以看到学生的姓名和学号属于数据部分，而 pNext 就是指针部分，用来保存下一个节点的地址。要向链表中添加一个节点时，操作的过程是怎样的呢？首先来看一组实例图，如图 11.14 所示。

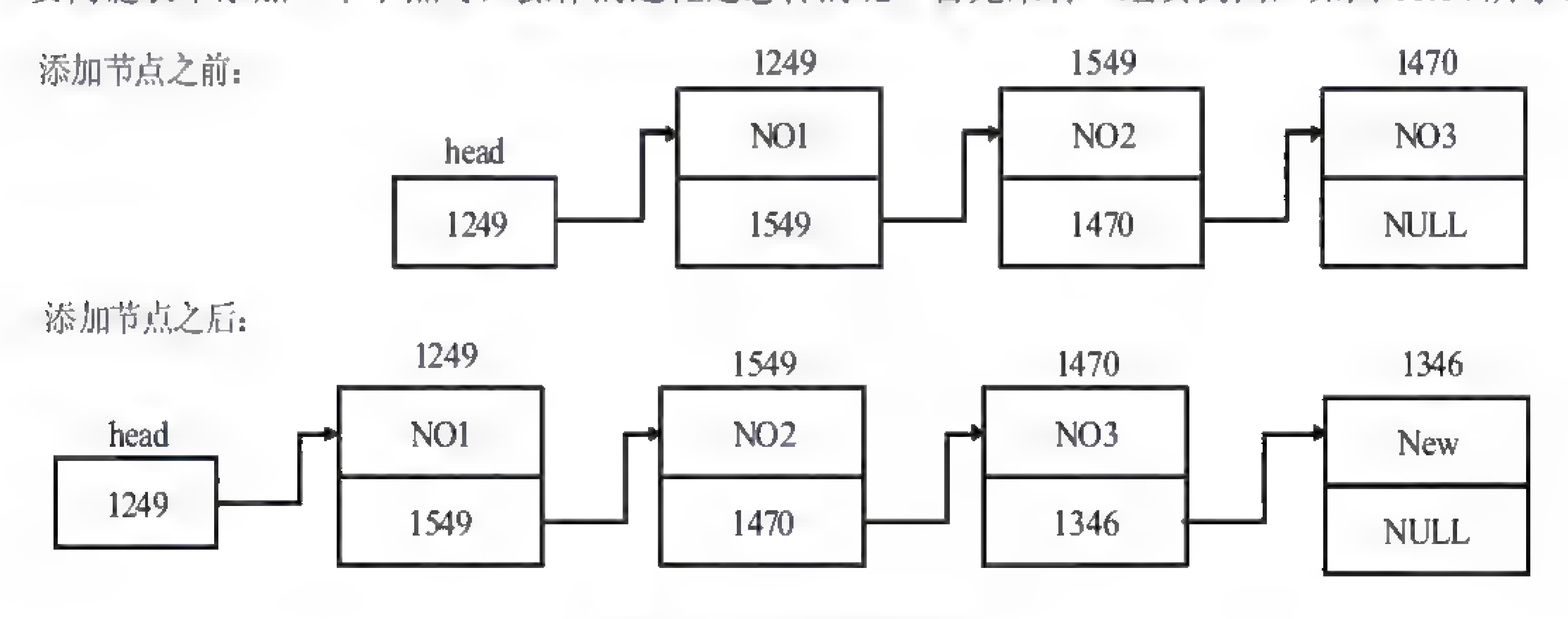


图 11.14　节点添加过程

当有新的节点要添加到链表中时，原来最后一个节点的指针将保存新添加的节点地址，而新节点的指针指向空（NULL），当添加完成后，新节点将成为链表中的最后一个节点。从添加节点的过程中就可以看出，不用担心链表的长度会超出范围。至于具体的代码内容将会在下面的小节中进行讲述。

11.5.2　创建动态链表

从本节开始讲解链表相关的具体操作，从对链表的概述中可以看出链表并不是一开始就设定好自身的大小，而是根据节点的多少而决定的，因此链表的创建过程是一个动态的创建过程。动态创建一

个节点时，要为其分配内存，在介绍如何创建链表前先来了解一些有关动态创建会使用的函数。

1．malloc 函数

malloc 函数的原型如下：

```
void *malloc(unsigned int size);
```

该函数的功能是在内存中动态地分配一块 size 大小的内存空间。malloc 函数会返回一个指针，该指针指向分配的内存空间，如果出现错误则返回 NULL。

2．calloc 函数

calloc 函数的原型如下：

```
void * calloc(unsigned n, unsigned size);
```

该函数的功能是在内存中动态分配 n 个长度为 size 的连续内存空间数组。calloc 函数会返回一个指针，该指针指向动态分配的连续内存空间地址。当分配空间错误时，返回 NULL。

3．free 函数

free 函数的原型如下：

```
void free(void *ptr);
```

该函数的功能是使用由指针 ptr 指向的内存区，使部分内存区能被其他变量使用。ptr 是最近一次调用 calloc 或 malloc 函数时返回的值。free 函数无返回值。

动态分配的相关函数已经介绍完了，现在开始介绍如何建立动态的链表。

所谓建立动态链表就是指在程序运行过程中从无到有地建立起一个链表，即一个一个地分配节点的内存空间，然后输入节点中的数据并建立节点间的相连关系。

例如，在链表概述中介绍过可以将一个班级里的学生作为链表中的节点，然后将所有学生的信息存放在链表结构中。

首先创建节点结构，表示每一个学生：

```
struct Student
{
	char cName[20];                     /*姓名*/
	int iNumber;                        /*学号*/
	struct Student* pNext;              /*指向下一个节点的指针*/
};
```

然后定义一个 Create 函数，用来创建列表。该函数将返回链表的头指针。

```
int iCount;                                 /*全局变量表示链表长度*/

struct Student* Create()
{
	struct Student* pHead=NULL;         /*初始化链表头指针为空*/
	struct Student* pEnd,*pNew;
```

```
    iCount=0;                                              /*初始化链表长度*/
    pEnd=pNew=(struct Student*)malloc(sizeof(struct Student));
    printf("please first enter Name ,then Number\n");
    scanf("%s",&pNew->cName);
    scanf("%d",&pNew->iNumber);
    while(pNew->iNumber!=0)
    {
        iCount++;
        if(iCount==1)
        {
            pNew->pNext=pHead;                             /*使得指向为空*/
            pEnd=pNew;                                     /*跟踪新加入的节点*/
            pHead=pNew;                                    /*头指针指向首节点*/
        }
        else
        {
            pNew->pNext=NULL;                              /*新节点的指针为空*/
            pEnd->pNext=pNew;                              /*原来的尾节点指向新节点*/
            pEnd=pNew;                                     /*pEnd 指向新节点*/
        }
        pNew=(struct Student*)malloc(sizeof(struct Student));  /*再次分配节点内存空间*/
        scanf("%s",&pNew->cName);
        scanf("%d",&pNew->iNumber);
    }
    free(pNew);                                            /*释放没有用到的空间*/
    return pHead;
}
```

Create 函数的功能是创建链表，在 Create 的外部可以看到一个整型的全局变量 iCount，这个变量的作用是表示链表中节点的数量。在 Create 函数中，首先定义需要用到的指针变量，pHead 用来表示头指针，pEnd 用来指向原来的尾节点，pNew 用来指向新创建的节点。

使用 malloc 函数分配内存，先用 pEnd 和 pNew 两个指针都指向第一个分配的内存，然后显示提示信息，先输出一个学生的姓名，再输入学生的学号。使用 while 语句进行判断，如果学号为 0，则不执行循环语句。

在 while 循环语句中，iCount++自加操作表示链表中节点的增加。然后要判断新加入的节点是否是第一次加入的节点，如果是第一次加入，则执行 if 语句块中的代码，否则应执行 else 语句块中的代码。

在 if 语句块中，因为第一次加入节点时其中没有节点，所以新节点即为首节点，也为最后一个节点，并且要将新加入的节点的指针指向 NULL，即为 pHead 指向。else 语句实现的是链表中已经有节点存在时的操作。首先将新节点 pNew 的指针指向 NULL，然后将原来最后一个节点的指针指向新节点，最后将 pEnd 指针指向最后一个节点。

一个节点创建完之后，要进行内存分配，然后向其中输入数据，通过 while 语句再次判断输入的数据是否符合节点的要求。当节点不符合要求时，执行下面的代码，即调用 free 函数将不符合要求的节点空间进行释放。

这样，一个链表就通过动态分配内存空间的方式创建完成了。

11.5.3　输出链表

链表已经被创建出来，构建数据结构就是为了使用它，以将保存的信息进行输出显示。接下来介绍如何将链表中的数据显示输出。

```
void Print(struct Student* pHead)
{
	struct Student *pTemp;                                  /*循环所用的临时指针*/
	int iIndex=1;                                           /*表示链表中节点的序号*/

	printf("----the List has %d members:----\n",iCount);    /*消息提示*/
	printf("\n");                                           /*换行*/
	pTemp=pHead;                                            /*指针得到首节点的地址*/

	while(pTemp!=NULL)
	{
		printf("the NO%d member is:\n",iIndex);
		printf("the name is: %s\n",pTemp->cName);       /*输出姓名*/
		printf("the number is: %d\n",pTemp->iNumber);   /*输出学号*/
		printf("\n");                                   /*输出换行*/
		pTemp=pTemp->pNext;                             /*移动临时指针到下一个节点*/
		iIndex++;                                       /*进行自加运算*/
	}
}
```

Print 函数用来将链表中的数据进行输出。在函数的参数中，pHead 表示一个链表的头节点。在函数中，定义一个临时的指针 pTemp 用来进行循环操作，定义一个整型变量表示链表中的节点序号，然后用临时指针变量 pTemp 保存首节点的地址。

使用 while 语句将所有节点中保存的数据都显示输出。其中每输出一个节点的内容后，就移动 pTemp 指针变量指向下一个节点的地址。当为最后一个节点时，所拥有的指针指向 NULL，此时循环结束。

【例 11.10】　创建链表并将数据输出。（**实例位置：资源包\TM\sl\11\10**）

根据上面介绍的有关链表的创建与输出操作，将这些代码整合到一起，编写一个包含学生信息的链表结构，并且将链表中的信息进行输出。

```
#include<stdio.h>
#include<stdlib.h>

struct Student
{
	char cName[20];                                         /*姓名*/
	int iNumber;                                            /*学号*/
	struct Student* pNext;                                  /*指向下一个节点的指针*/
};
```

```
int iCount;                                              /*全局变量表示链表长度*/

struct Student* Create()
{
    struct Student* pHead=NULL;                          /*初始化链表头指针为空*/
    struct Student* pEnd,*pNew;
    iCount=0;                                            /*初始化链表长度*/
    pEnd=pNew=(struct Student*)malloc(sizeof(struct Student));
    printf("please first enter Name ,then Number\n");
    scanf("%s",&pNew->cName);
    scanf("%d",&pNew->iNumber);
    while(pNew->iNumber!=0)
    {
        iCount++;
        if(iCount==1)
        {
            pNew->pNext=pHead;                           /*使得指向为空*/
            pEnd=pNew;                                   /*跟踪新加入的节点*/
            pHead=pNew;                                  /*头指针指向首节点*/
        }
        else
        {
            pNew->pNext=NULL;                            /*新节点的指针为空*/
            pEnd->pNext=pNew;                            /*原来的尾节点指向新节点*/
            pEnd=pNew;                                   /*pEnd 指向新节点*/
        }
        pNew=(struct Student*)malloc(sizeof(struct Student));  /*再次分配节点内存空间*/
        scanf("%s",&pNew->cName);
        scanf("%d",&pNew->iNumber);
    }
    free(pNew);                                          /*释放没有用到的空间*/
    return pHead;
}

void Print(struct Student* pHead)
{
    struct Student *pTemp;                               /*循环所用的临时指针*/
    int iIndex=1;                                        /*表示链表中节点的序号*/

    printf("----the List has %d members:----\n",iCount);  /*消息提示*/
    printf("\n");                                        /*换行*/
    pTemp=pHead;                                         /*指针得到首节点的地址*/

    while(pTemp!=NULL)
    {
        printf("the NO%d member is:\n",iIndex);
        printf("the name is: %s\n",pTemp->cName);        /*输出姓名*/
        printf("the number is: %d\n",pTemp->iNumber);    /*输出学号*/
        printf("\n");                                    /*输出换行*/
        pTemp=pTemp->pNext;                              /*移动临时指针到下一个节点*/
```

```
            iIndex++;                                    /*进行自加运算*/
        }
}

int main()
{
        struct Student* pHead;                           /*定义头节点*/
        pHead=Create();                                  /*创建节点*/
        Print(pHead);                                    /*输出链表*/
        return 0;                                        /*程序结束*/
}
```

在 main 函数中，先定义一个头节点指针 pHead，然后调用 Create 函数创建链表，并将链表的头节点返回给 pHead 指针变量。利用得到的头节点 pHead 作为 Print 函数的参数。

运行程序，显示效果如图 11.15 所示。

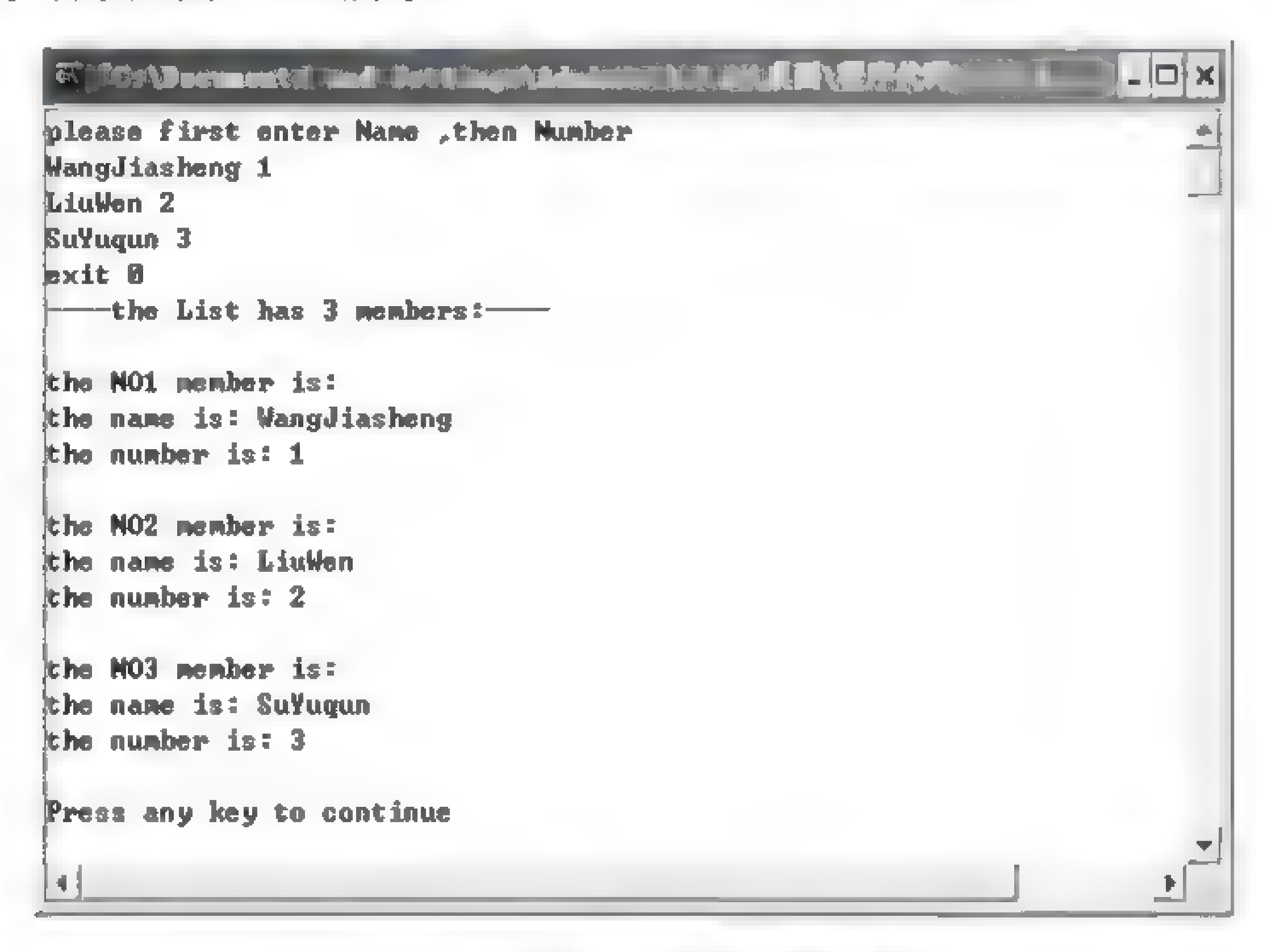


图 11.15　创建链表并将数据输出

11.6　链表相关操作



本节将对链表的功能进行完善，使其具有插入、删除节点的功能。这些操作都是在 11.5 节中所声明的结构和链表的基础上添加的。

11.6.1　链表的插入操作

链表的插入操作可以在链表的头节点位置进行，也可以在某个节点的位置进行，或者可以像创建结构时在链表的后面添加节点。这 3 种插入操作的思路都是一样的。下面主要介绍第一种插入方式，在链表的头节点位置插入节点，如图 11.16 所示。

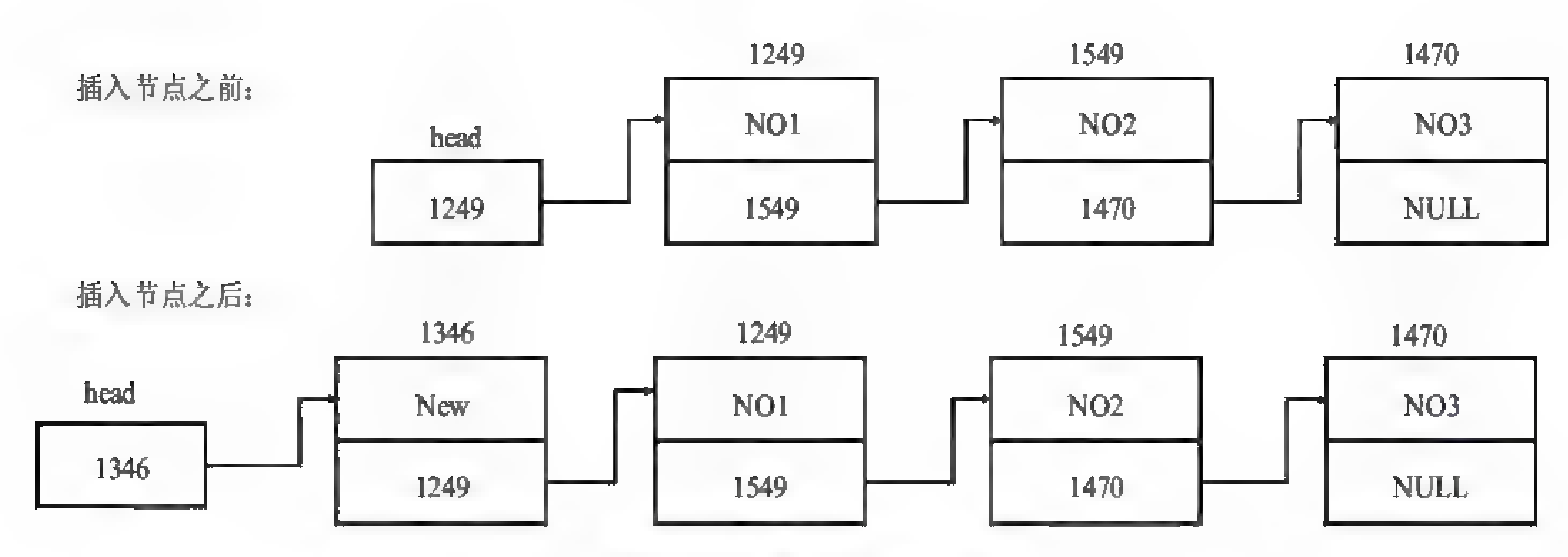


图 11.16　插入节点操作

插入节点的过程就如手拉手的小朋友连成一条线，这时又来了一个小朋友，他要站在老师和一个小朋友的中间，那么老师就要放开原来的小朋友，拉住新加入的小朋友，这个新加入的小朋友就拉住原来的那个小朋友。这样，这条连成的线还是连在一起。

设计一个函数用来向链表中添加节点：

```
struct Student* Insert(struct Student* pHead)
{
	struct Student* pNew;                                    /*指向新分配的空间*/
	printf("----Insert member at first----\n");             /*提示信息*/
	pNew=(struct Student*)malloc(sizeof(struct Student));   /*分配内存空间，并返回指向该内存空间的指针*/

	scanf("%s",&pNew->cName);
	scanf("%d",&pNew->iNumber);

	pNew->pNext=pHead;                                       /*新节点指针指向原来的首节点*/
	pHead=pNew;                                              /*头指针指向新节点*/
	iCount++;                                                /*增加链表节点数量*/
	return pHead;                                            /*返回头指针*/
}
```

在代码中，为要插入的新节点分配内存，然后向新节点中输入数据，这样一个节点就创建完成了。接下来就是将这个节点插入链表中。首先将新节点的指针指向原来的首节点，保存首节点的地址。然后将头指针指向新节点，这样就完成了节点的连接操作，最后增加链表的节点数量。

修改 main 函数的代码，加入添加节点操作：

```
int main()
{
	struct Student* pHead;                                   /*定义头节点*/
	pHead=Create();                                          /*创建节点*/
	pHead=Insert(pHead);                                     /*插入节点*/
	Print(pHead);                                            /*输出链表*/
	return 0;                                                /*程序结束*/
}
```

使用 Insert 函数返回新的头指针，运行程序，显示效果如图 11.17 所示。

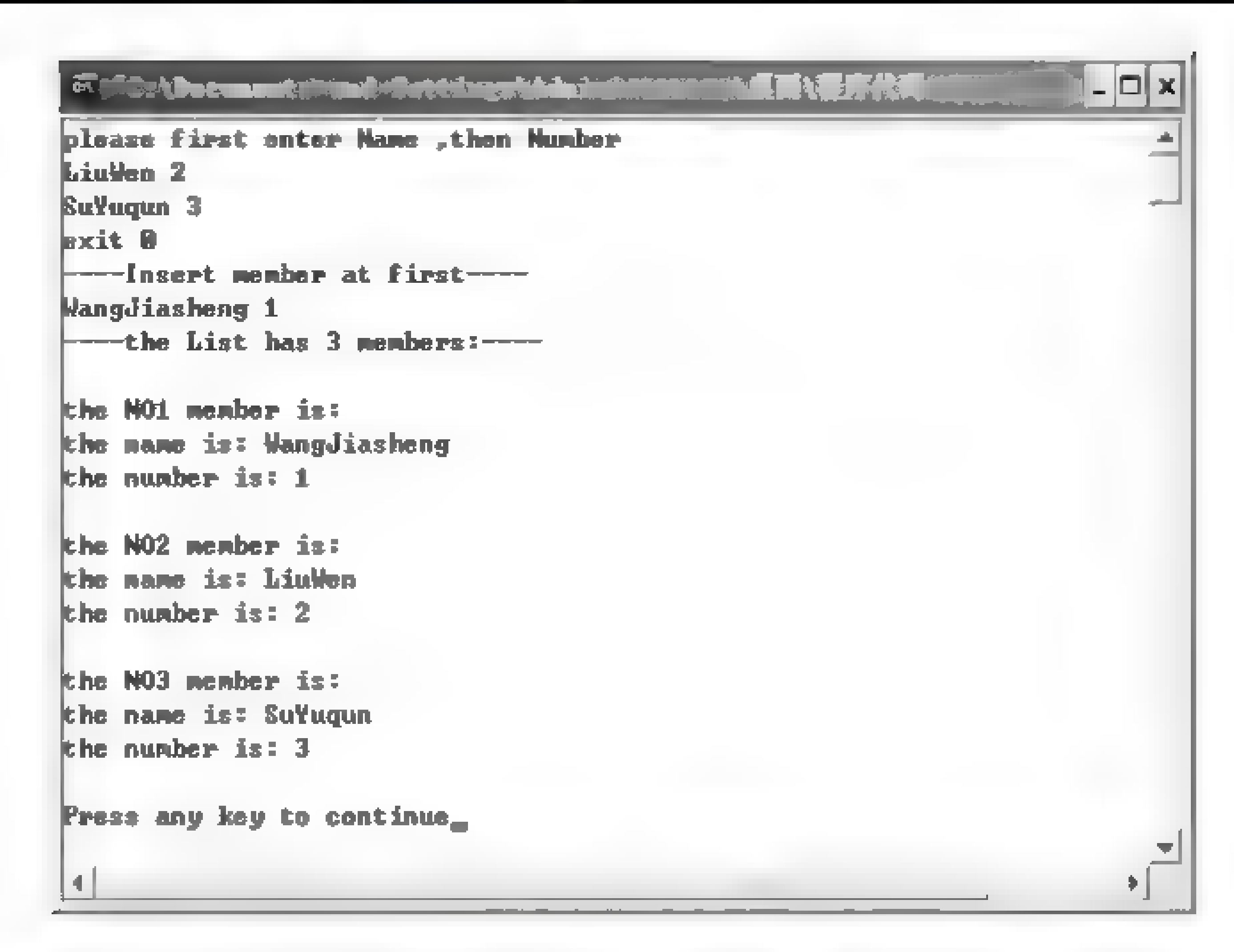


图 11.17　链表插入操作

11.6.2　链表的删除操作

之前的操作都是向链表中添加节点，当希望删除链表中的节点时，应该怎么办呢？还是通过前文中小朋友手拉手的比喻进行理解。例如，队伍中的一个小朋友想离开队伍了，并且这个队伍不会断开的方法是只需他两边的小朋友将手拉起来就可以了。

例如，在一个链表中删除其中的一点，如图 11.18 所示。

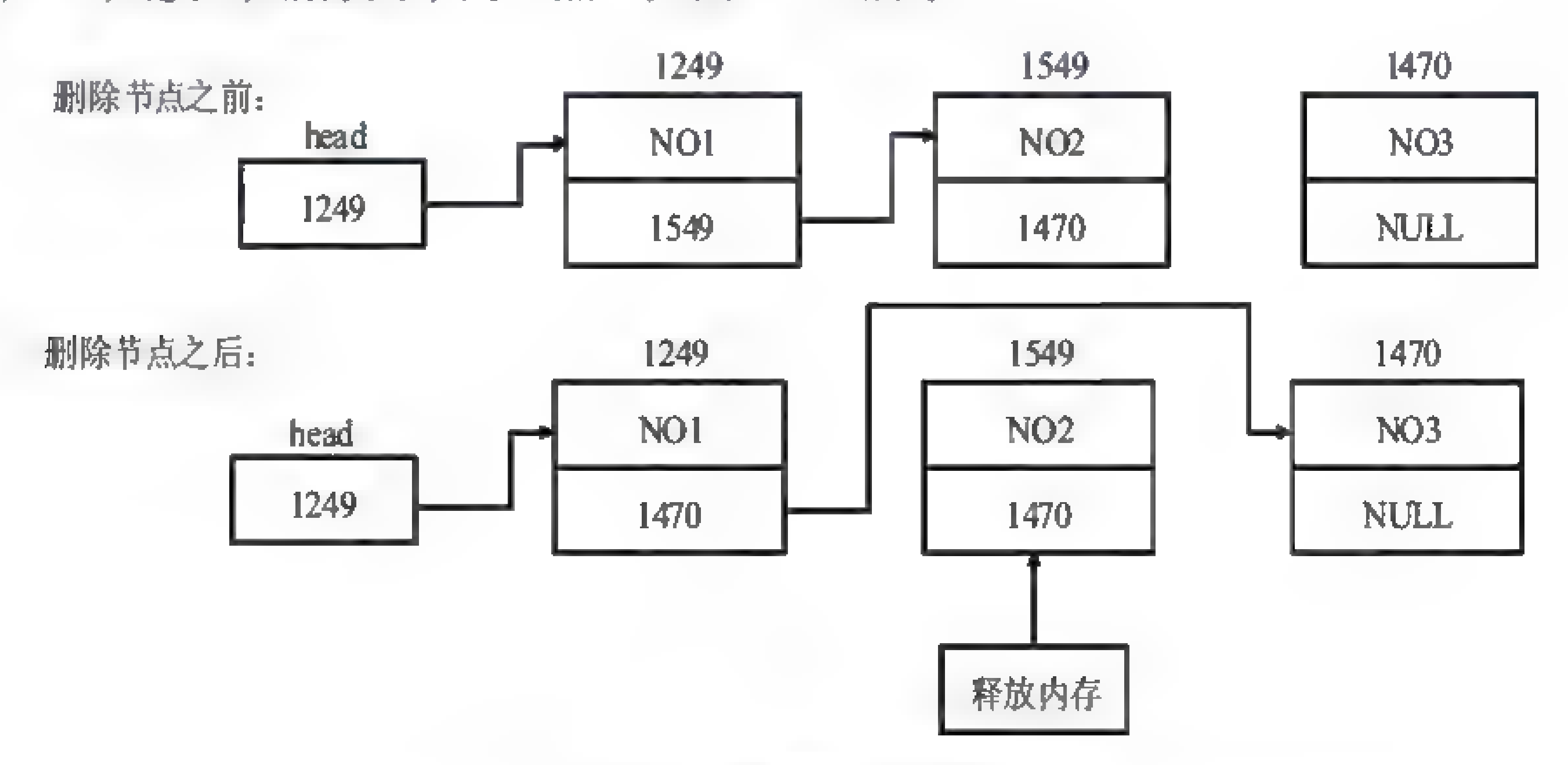


图 11.18　删除节点操作

通过图 11.18 可以发现，要删除一个节点，首先要找到这个节点的位置，例如图中的 NO2 节点。然后将 NO1 节点的指针指向 NO3 节点，最后将 NO2 节点的内存空间释放掉，这样就完成了节点的删除操作。

根据这个思路可以编写删除链表节点操作的函数如下：

```
void Delete(struct Student* pHead,int iIndex)          /*pHead 表示头节点，iIndex 表示要删除的节点下标*/
{
    int i;                                             /*控制循环变量*/
    struct Student* pTemp;                             /*临时指针*/
    struct Student* pPre;                              /*表示要删除节点前的节点*/
```

```
    pTemp=pHead;                                  /*得到头节点*/
    pPre=pTemp;

    printf("----delete NO%d member----\n",iIndex);  /*提示信息*/
    for(i=1;i<iIndex;i++)                         /*for 循环使得 pTemp 指向要删除的节点*/
    {
        pPre=pTemp;
        pTemp=pTemp->pNext;
    }
    pPre->pNext=pTemp->pNext;                     /*连接删除节点两边的节点*/
    free(pTemp);                                  /*释放掉要删除节点的内存空间*/
    iCount--;                                     /*减少链表中的元素个数*/
}
```

为 Delete 函数传递两个参数，pHead 表示链表的头指针，iIndex 表示要删除节点在链表中的位置。定义整型变量 i 用来控制循环的次数，然后定义两个指针，分别用来表示要删除的节点和这个节点之前的节点。

输出一行提示信息，表示要进行删除操作。之后利用 for 语句进行循环操作，找到要删除的节点，并使用 pTemp 保存要删除节点的地址，pPre 保存前一个节点的地址。找到要删除的节点后，连接删除节点两边的节点，并使用 free 函数将 pTemp 指向的内存空间进行释放。

接下来在 main 函数中添加代码执行删除操作，将链表中的第二个节点进行删除。

```
int main()
{
    struct Student* pHead;                        /*定义头节点*/
    pHead=Create();                               /*创建节点*/
    pHead=Insert(pHead);                          /*插入节点*/
    Delete(pHead,2);                              /*删除第二个节点的操作*/
    Print(pHead);                                 /*输出链表*/
    return 0;                                     /*程序结束*/
}
```

运行程序，通过显示的结果可以看到第二个节点中的数据已被删除，显示效果如图 11.19 所示。

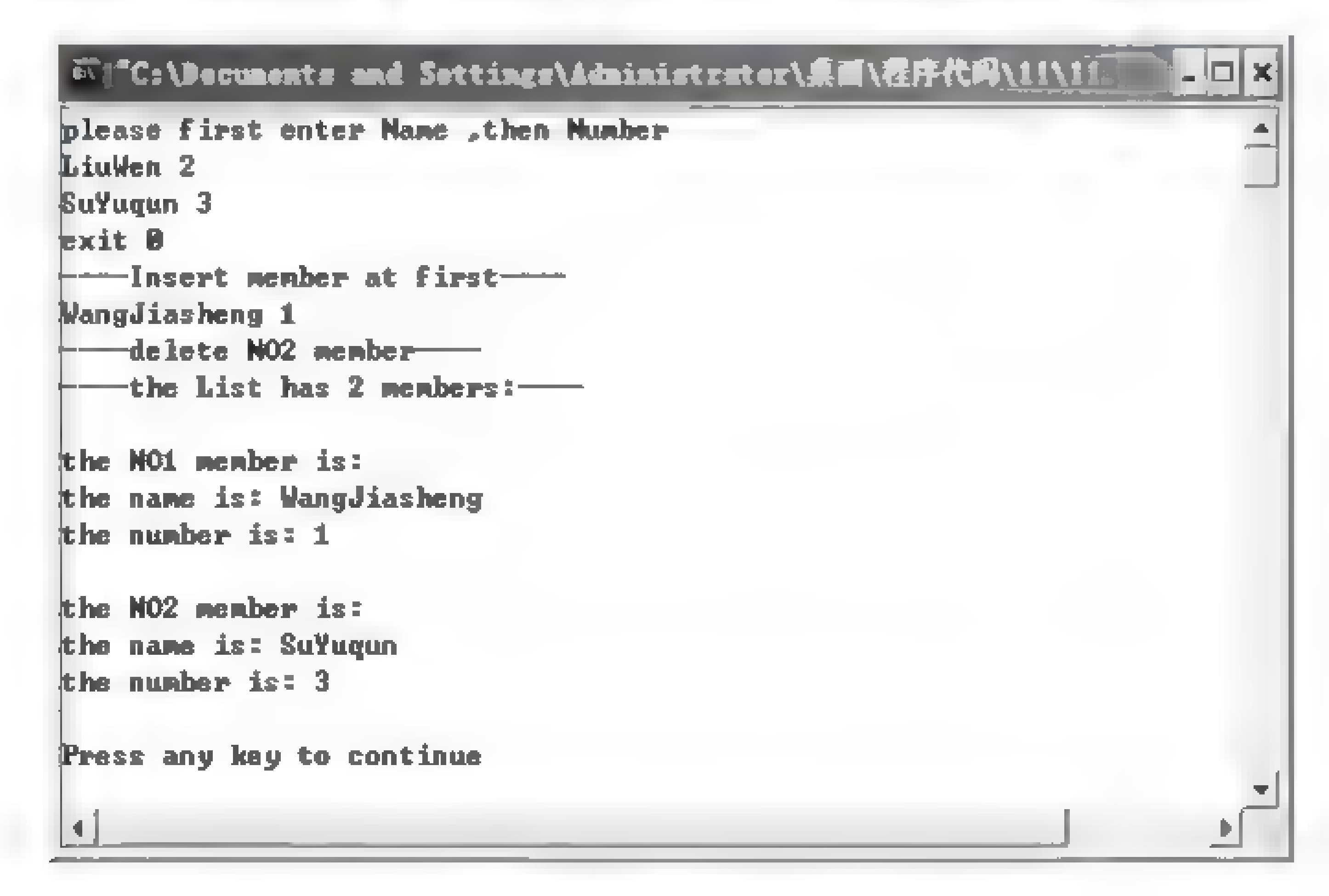


图 11.19　删除节点操作

有关链表的操作就讲解到这里。为了方便读者阅读程序，这里将有关链表操作的完整程序给出，希望读者能从整体上对链表有一个清晰的理解。

【例 11.11】　完整的链表操作代码。(**实例位置：资源包\TM\sl\11\11**)

```
#include<stdio.h>
#include<stdlib.h>

struct Student
{
	char cName[20];                                                  /*姓名*/
	int iNumber;                                                     /*学号*/
	struct Student* pNext;                                           /*指向下一个节点的指针*/
};

int iCount;                                                          /*全局变量表示链表长度*/

struct Student* Create()
{
	struct Student* pHead=NULL;                                      /*初始化链表，头指针为空*/
	struct Student* pEnd,*pNew;
	iCount=0;                                                        /*初始化链表长度*/
	pEnd=pNew=(struct Student*)malloc(sizeof(struct Student));
	printf("please first enter Name ,then Number\n");
	scanf("%s",&pNew->cName);
	scanf("%d",&pNew->iNumber);
	while(pNew->iNumber!=0)
	{
		iCount++;
		if(iCount==1)
		{
			pNew->pNext=pHead;                                       /*使得指向为空*/
			pEnd=pNew;                                               /*跟踪新加入的节点*/
			pHead=pNew;                                              /*头指针指向首节点*/
		}
		else
		{
			pNew->pNext=NULL;                                        /*新节点的指针为空*/
			pEnd->pNext=pNew;                                        /*原来的尾节点指向新节点*/
			pEnd=pNew;                                               /*pEnd 指向新节点*/
		}
		pNew=(struct Student*)malloc(sizeof(struct Student));        /*再次分配节点内存空间*/
		scanf("%s",&pNew->cName);
		scanf("%d",&pNew->iNumber);
	}
	free(pNew);                                                      /*释放没有用到的空间*/
	return pHead;
}

void Print(struct Student* pHead)
```

```
{
    struct Student *pTemp;                          /*循环所用的临时指针*/
    int iIndex=1;                                   /*表示链表中节点的序号*/

    printf("----the List has %d members:----\n",iCount);  /*消息提示*/
    printf("\n");                                   /*换行*/
    pTemp=pHead;                                    /*指针得到首节点的地址*/

    while(pTemp!=NULL)
    {
        printf("the NO%d member is:\n",iIndex);
        printf("the name is: %s\n",pTemp->cName);   /*输出姓名*/
        printf("the number is: %d\n",pTemp->iNumber); /*输出学号*/
        printf("\n");                               /*输出换行*/
        pTemp=pTemp->pNext;                         /*移动临时指针到下一个节点*/
        iIndex++;                                   /*进行自加运算*/
    }
}

struct Student* Insert(struct Student* pHead)
{
    struct Student* pNew;                           /*指向新分配的空间*/
    printf("----Insert member at first----\n");    /*提示信息*/
    pNew=(struct Student*)malloc(sizeof(struct Student)); /*分配内存空间，并返回指向该内存空间的指针*/

    scanf("%s",&pNew->cName);
    scanf("%d",&pNew->iNumber);

    pNew->pNext=pHead;                              /*新节点指针指向原来的首节点*/
    pHead=pNew;                                     /*头指针指向新节点*/
    iCount++;                                       /*增加链表节点数量*/
    return pHead;
}

void Delete(struct Student* pHead,int iIndex)       /*pHead 表示头节点，iIndex 表示要删除的节点下标*/
{
    int i;                                          /*控制循环变量*/
    struct Student* pTemp;                          /*临时指针*/
    struct Student* pPre;                           /*表示要删除节点前的节点*/
    pTemp=pHead;                                    /*得到头节点*/
    pPre=pTemp;

    printf("----delete NO%d member----\n",iIndex);  /*提示信息*/
    for(i=1;i<iIndex;i++)                           /*for 循环使得 pTemp 指向要删除的节点*/
    {
        pPre=pTemp;
        pTemp=pTemp->pNext;
    }
    pPre->pNext=pTemp->pNext;                       /*连接删除节点两边的节点*/
```

```
        free(pTemp);                          /*释放要删除节点的内存空间*/
        iCount--;                             /*减少链表中的元素个数*/
}

int main()
{
        struct Student* pHead;                /*定义头节点*/
        pHead=Create();                       /*创建节点*/
        pHead=Insert(pHead);                  /*插入节点*/
        Delete(pHead,2);                      /*删除第二个节点*/
        Print(pHead);                         /*输出链表*/
        return 0;                             /*程序结束*/
}
```

11.7　共　用　体


视频讲解

共用体看起来很像结构体，只不过关键字由 struct 变成了 union。共用体和结构体的区别在于：结构体定义了一个由多个数据成员组成的特殊类型，而共用体定义了一块为所有数据成员共享的内存。

11.7.1　共用体的概念

共用体也称为联合体，它使几种不同类型的变量存放到同一段内存单元中。所以共用体在同一时刻只能有一个值，它属于某一个数据成员。由于所有成员位于同一块内存，因此共用体的大小就等于最大成员的大小。

定义共用体的类型变量的一般形式如下：

```
union 共用体名
{
        成员列表
}变量列表;
```

例如，下面的代码定义一个共用体，包括的数据成员有整型、字符型和实型：

```
union DataUnion
{
        int iInt;
        char cChar;
        float fFloat;
}variable;                                    /*定义共用体变量*/
```

其中，variable 为定义的共用体变量，而 union DataUnion 是共用体类型。还可以像结构体那样，将类型的声明和变量定义分开：

```
union DataUnion variable;
```

可以看到，共用体定义变量的方式与结构体定义变量的方式很相似。不过一定要注意的是，结构体变量的大小是其所包括的所有数据成员大小的总和，其中每个成员分别占有自己的内存单元；而共用体的大小为所包含数据成员中最大内存长度的大小。例如，上面定义的共用体变量 variable 的大小就与 float 类型的大小相等。

11.7.2 共用体变量的引用

共用体变量定义完成后，就可以引用其中的成员数据。引用的一般形式如下：

```
共用体变量.成员名;
```

例如，引用前面定义的 variable 变量中的成员数据的方法：

```
variable.iInt;
variable.cChar;
variable.fFloat;
```

注意

不能直接引用共用体变量，如“printf("%d",variable);”。

【例 11.12】 使用共用体变量。（**实例位置：资源包\TM\sl\11\12**）

在本实例中定义共用体变量，通过定义的显示函数，引用共用体中的数据成员。

```
#include<stdio.h>

union DataUnion                              /*声明共用体类型*/
{
    int iInt;                                /*成员变量*/
    char cChar;
};

int main()
{
    union DataUnion Union;                   /*定义共用体变量*/
    Union.iInt=97;                           /*为共用体变量中的成员赋值*/
    printf("iInt: %d\n",Union.iInt);         /*输出成员变量数据*/
    printf("cChar: %c\n",Union.cChar);
    Union.cChar='A';                         /*改变成员的数据*/
    printf("iInt: %d\n",Union.iInt);         /*输出成员变量数据*/
    printf("cChar: %c\n",Union.cChar);
    return 0;
}
```

在程序中改变共用体的一个成员，其他成员也会随之改变。当给某个特定的成员进行赋值时，其他成员的值也会具有一致的含义，这是因为它们的值的每一个二进制位都被新值所覆盖。

运行程序，显示效果如图 11.20 所示。

```
iInt: 97
cChar: a
iInt: 65
cChar: A
Press any key to continue
```

图 11.20　使用共用体变量

11.7.3　共用体变量的初始化

在定义共用体变量时，可以同时对变量进行初始化操作。初始化的值放在一对大括号中。

注意

对共用体变量初始化时，只需要一个初始化值就足够了，其类型必须和共用体的第一个成员的类型相一致。

【例 11.13】　共用体变量的初始化。（**实例位置：资源包\TM\sl\11\13**）

在本实例中，在定义共用体变量的同时进行初始化操作，并将引用变量的值输出。

```
#include<stdio.h>

union DataUnion                              /*声明共用体类型*/
{
    int iInt;                                /*成员变量*/
    char cChar;
};

int main()
{
    union DataUnion Union={97};              /*定义共用体变量，并进行初始化*/
    printf("iInt: %d\n",Union.iInt);         /*输出成员变量数据*/
    printf("cChar: %c\n",Union.cChar);
    return 0;
}
```

说明

如果共用体的第一个成员是一个结构体类型，则初始化值中可以包含多个用于初始化该结构的表达式。

运行程序，显示效果如图 11.21 所示。



图 11.21　共用体变量的初始化

11.7.4 共用体类型的数据特点

在使用结构体类型时，需要注意以下特点：

☑ 同一内存段可以用来存放几种不同类型的成员，但是每次只能存放其中一种，而不能同时存放所有的类型。也就是说在共用体中，只有一个成员起作用，其他成员不起作用。

☑ 共用体变量中起作用的成员是最后一次存放的成员。在存入一个新的成员后，原有的成员就失去作用。

☑ 共用体变量的地址和它的各成员的地址是一样的。

☑ 不能对共用体变量名赋值，也不能企图引用变量名来得到一个值。


视频讲解

11.8 枚举类型

利用关键字 enum 可以声明枚举类型，这也是一种数据类型。使用该类型可以定义枚举类型变量，一个枚举变量包含一组相关的标识符，其中每个标识符都对应一个整数值，称为枚举常量。

例如，定义一个枚举类型变量，其中每个标识符都对应一个整数值：

```
enum Colors(Red,Green,Blue);
```

Colors 就是定义的枚举类型变量，在括号中的第一个标识符对应着数值 0，第二个对应于 1，依此类推。

注意

每个标识符都必须是唯一的，而且不能采用关键字或当前作用域内其他相同的标识符名。

在定义枚举类型的变量时，可以为某个特定的标识符指定其对应的整型值，紧随其后的标识符对应的值依次加 1。例如：

```
enum Colors(Red=1,Green,Blue);
```

这样的话，Red 的值为 1，Green 为 2，Blue 为 3。

【例 11.14】 使用枚举类型。（**实例位置：资源包\TM\sl\11\14**）

在本实例中，通过定义枚举类型观察其使用方式，其中每个枚举常量在声明的作用域内都可以看作一个新的数据类型。

```
#include<stdio.h>

enum Color{Red=1,Blue,Green} color;                /*定义枚举变量，并初始化*/
int main()
{
    int icolor;                                     /*定义整型变量*/
```

```
	scanf("%d",&icolor);                    /*输入数据*/
	switch(icolor)                          /*判断 icolor 值*/
	{
	case Red:                               /*枚举常量，Red 表示 1*/
		printf("the choice is Red\n");
		break;
	case Blue:                              /*枚举常量，Blue 表示 2*/
		printf("the choice is Blue\n");
		break;
	case Green:                             /*枚举常量，Green 表示 3*/
		printf("the choice is Green\n");
		break;
	default:
		printf("???\n");
		break;
	}
	return 0;
}
```

在程序中首先定义了一个枚举变量。在初始化时，为第一个枚举常量赋值为 1，这样 Red 赋值为 1 后，之后的枚举常量就会依次加 1。通过 switch 语句判断输入的数据与这些标识符是否符合，然后执行 case 语句中的操作。

运行程序，显示效果如图 11.22 所示。



图 11.22　使用枚举类型

11.9　小　　结

本章先介绍了有关结构体的内容，编程人员可以通过结构定义符合要求的结构类型。之后介绍了结构体以数组方式定义，指向结构体的指针，以及包含结构的结构的情况。

学习完如何构建结构体后，接下来介绍了一种常见的数据结构——链表。其中讲解了有关链表的创建过程，介绍如何动态分配内存空间。而链表的插入、删除、输出操作，应用了之前学习的结构体的知识。

本章最后讲解了有关共用体和枚举类型这两方面的内容，共用体的一般形式与结构体类似，但它们之间还是存在区别：结构体的大小是所有成员数据大小的总和，而共用体的大小与成员数据中最大的成员相同；而枚举类型与结构体、共用体都不同，枚举类型的元素是常量。

11.10 实践与练习

1．设计一个候选人的选票程序。假设有3个候选人，每一次输入要选择的候选人姓名，最后输出每个人的得票结果。**（答案位置：资源包\TM\sl\11\15）**

2．设计一个程序，可以存放学校中所有人员（包括学生和老师）的数据。学生的数据包括身份、姓名、编号、性别和班级；老师的数据包括身份、姓名、编号、性别和职务。（提示：由于老师和学生两者数据中只有一项是不相同的，所以可以使用共用体，这样设计一个结构体类型就可以满足设计要求）**（答案位置：资源包\TM\sl\11\16）**

第12章

位运算

（视频讲解：36分钟）

C语言可用来代替汇编语言，完成大部分编程工作。也就是说，C语言能支持汇编语言做大部分的运算，这是因为C语言完全支持按位运算。这也是C语言的一个特点，这个特点使C语言的应用更加广泛。

通过阅读本章，您可以：

- 掌握6种位运算符
- 掌握实现循环移位的方法
- 了解位段的相关内容



视频讲解

12.1 位与字节

在前面章节中讲过数据在内存中是以二进制的形式存放的，下面将具体介绍位与字节之间的关系。

位是计算机存储数据的最小单位。一个二进制位可以表示两种状态（0 和 1），多个二进制位组合起来便可表示多种信息。

一个字节通常是由 8 位二进制数组成，当然有的计算机系统是由 16 位组成，本书中提到的一个字节指的是由 8 位二进制组成的。

因为本书中所使用的运行环境是 Visual C++ 6.0，所以定义一个基本整型数据，它在内存中占 4 个字节，也就是 32 位；如果定义一个字符型，则在内存中占一个字节，也就是 8 位。不同的数据类型占用的字节数不同，因此占用的二进制位数也不同。



视频讲解

12.2 位运算操作符

C 语言既具有高级语言的特点，又具有低级语言的功能，C 语言和其他语言的区别是完全支持按位运算，而且也能像汇编语言一样用来编写系统程序。前面讲过的都是以字节为基本单位进行运算的，本节将介绍如何在位一级进行运算，按位运算也就是对字节或字中的实际位进行检测、设置或移位。表 12.1 所示为 C 语言提供的位运算符。

表 12.1 位运算符

位运算符	含义	位运算符	含义
&	按位与	^	按位异或
	按位或	<<	左移
~	取反	>>	右移

12.2.1 “与”运算符

按位“与”运算符“&”是双目运算符，功能是使参与运算的两数各对应的二进位相“与”。只有对应的两个二进位均为 1 时，结果才为 1，否则为 0，如表 12.2 所示。

表 12.2 “与”运算符

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

例如，89&38 的计算过程如下：

```
          0000000001011001    十进制数 89
（&）
          0000000000100110    十进制数 38
——————————————————————————————
          0000000000000000    十进制数 0
```

通过上面的运算可发现：按位“与”的一个用途就是清零。要将原数中为 1 的位置为 0，只需使与其进行“与”操作的数所对应的位置为 0 便可实现清零操作。

“与”操作的另一个用途就是取特定位，可以通过“与”的方式取一个数中的某些指定位。如果取 22 的后 5 位，则要与后 5 位均是 1 的数相“与”。同样，要取后 4 位，就与后 4 位都是 1 的数相“与”即可。

【例 12.1】　任意输入两个数，分别赋予 a 和 b，计算 a&b 的值。（**实例位置：资源包\TM\sl\12\1**）

```
#include<stdio.h>
main()
{
    unsigned result;                              /*定义无符号变量*/
    int a, b;
    printf("please input a:");
    scanf("%d",&a);
    printf("please input b:");
    scanf("%d",&b);
    printf("a=%d,b=%d", a, b);
    result = a&b;                                 /*计算“与”运算的结果*/
    printf("\na&b=%u\n", result);
}
```

程序运行结果如图 12.1 所示。

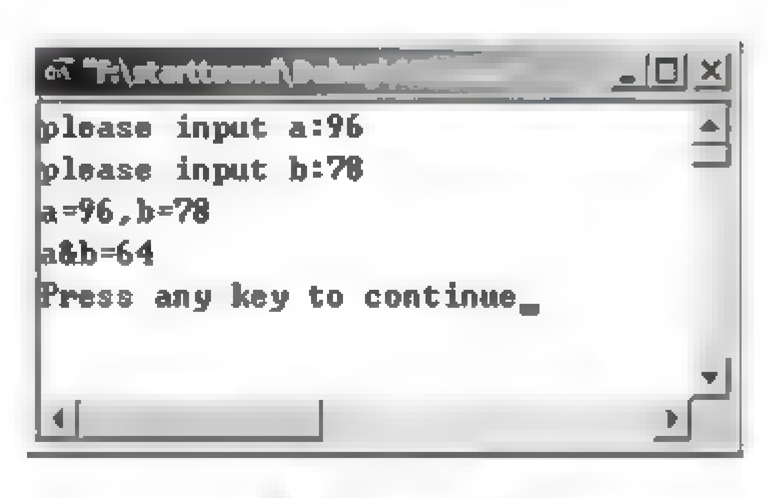


图 12.1　a&b

例 12.1 的计算过程如下：

```
          0000000001100000    十进制数 96
（&）
          0000000001001110    十进制数 78
——————————————————————————————
          0000000001000000    十进制数 64
```

12.2.2 “或”运算符

按位“或”运算符“|”是双目运算符，功能是使参与运算的两数各对应的二进位相“或”。只要对应的两个二进位有一个为 1，结果位就为 1，如表 12.3 所示。

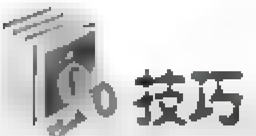
表 12.3 “或”运算符

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

例如，17|31 的计算过程如下：

```
     0000000000010001    十进制数 17
(|)
     0000000000011111    十进制数 31
―――――――――――――――――――――――
     0000000000011111    十进制数 31
```

从上式可以发现，十进制数 17 的二进制数的后 5 位是 10001，而十进制数 31 对应的二进制数的后 5 位是 11111，将这两个数执行“或”运算之后得到的结果是 31，也就是将 17 的二进制数的后 5 位中是 0 的位变成了 1。因此，可以总结出这样一个规律：要想使一个数的后 6 位全为 1，只需和 63 按位“或”即可；同理，若要使后 5 位全为 1，只需和 31 按位“或”即可，其他依此类推。


技巧

如果要将某几位为 1，只需与这几位是 1 的数执行“或”操作便可。

【例 12.2】 任意输入两个数，分别赋予 a 和 b，计算 a|b 的值。（**实例位置：资源包\TM\sl\12\2**）

```
#include<stdio.h>
main()
{
    unsigned result;                          /*定义无符号变量*/
    int a, b;
    printf("please input a:");
    scanf("%d",&a);
    printf("please input b:");
    scanf("%d",&b);
    printf("a=%d,b=%d", a, b);
    result = a|b;                             /*计算或运算的结果*/
    printf("\na|b=%u\n", result);
}
```

程序运行结果如图 12.2 所示。

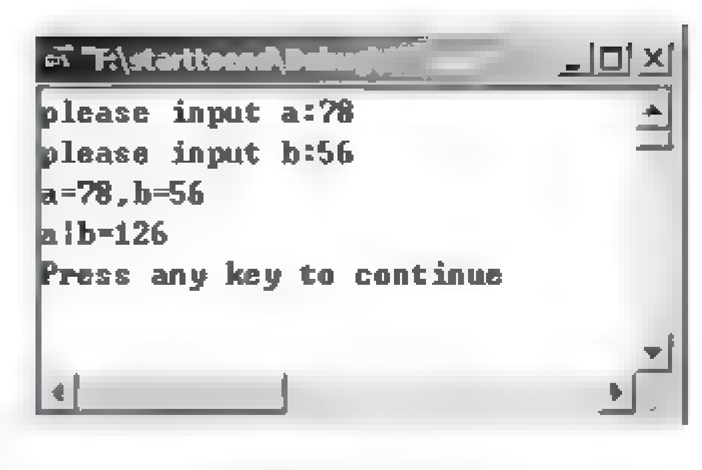


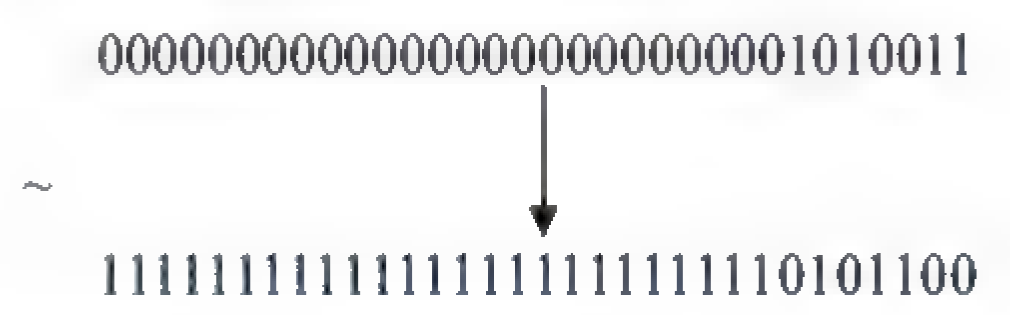
图 12.2　a|b

例 12.2 的计算过程如下（为了方便观察，这里只给出每个数据的后 16 位）：

```
      0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0
(|)
      0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0
    ----------------------------------
      0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
```

12.2.3　“取反”运算符

“取反”运算符“~”为单目运算符，具有右结合性。其功能是对参与运算的数的各二进位按位求反，即将 0 变成 1，1 变成 0。如~86 是对 86 进行按位求反：



注意

在进行“取反”运算的过程中，切不可简单地认为一个数取反后的结果就是该数的相反数（即~25 的值是-25），这是错误的。

【例 12.3】　输入一个数赋予变量 a，计算~a 的值。（**实例位置：资源包\TM\sl\12\3**）

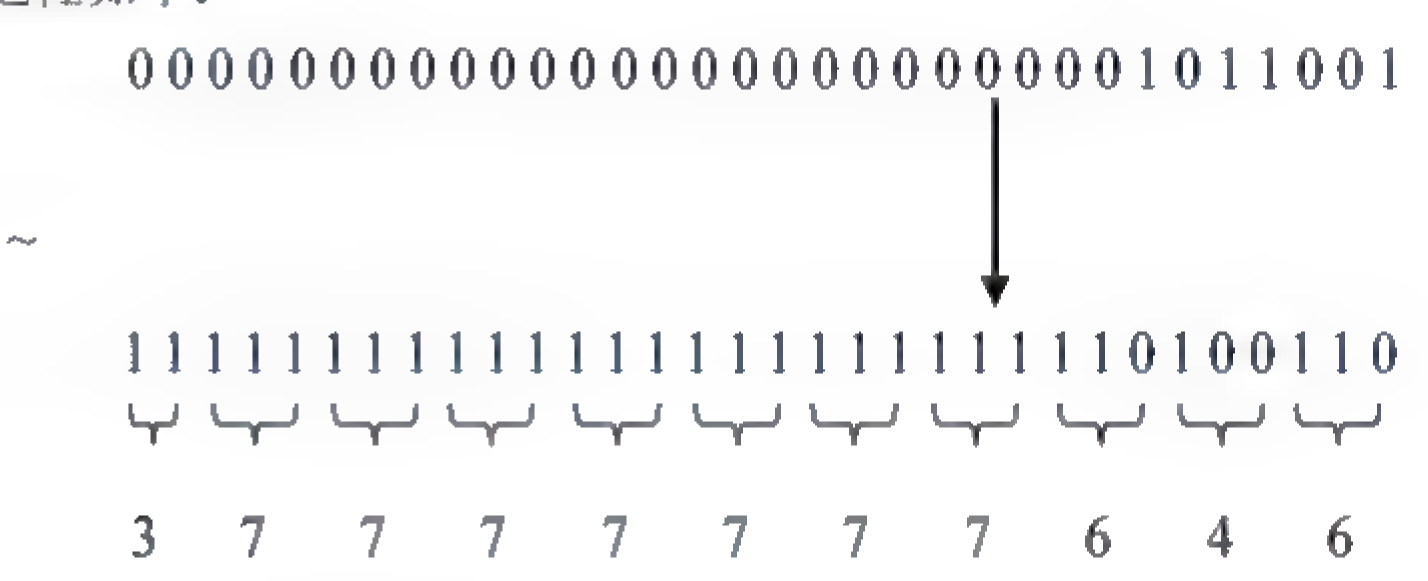
```
#include<stdio.h>
main()
{
    unsigned result;                                /*定义无符号变量*/
    int a;
    printf("please input a:");
    scanf("%d",&a);
    printf("a=%d", a);
    result = ~a;                                    /*求 a 的反*/
    printf("\n~a=%o\n", result);
}
```

程序运行结果如图 12.3 所示。



图 12.3　~a

例 12.3 的执行过程如下：



注意

例 12.3 最后是以八进制的形式输出的。

12.2.4　“异或”运算符

按位“异或”运算符“^”是双目运算符。其功能是使参与运算的两数各对应的二进位相“异或”，当对应的两个二进位数相异时结果为 1，否则结果为 0，如表 12.4 所示。

表 12.4　“异或”运算符

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

例如，107^127 的算式：

```
  0000000001101011
^
  0000000001111111
 ------------------
  0000000000010100
```

从上面算式可以看出，“异或”操作的一个主要用途就是能使特定的位翻转。例如，如果要将 107 的后 7 位翻转，只需与一个后 7 位都是 1 的数进行“异或”操作即可。

“异或”操作的另一个主要用途，就是在不使用临时变量的情况下实现两个变量值的互换。

例如，x=9，y=4，将 x 和 y 的值互换可用如下方法实现：

```
x=x^y;
y=y^x;
x=x^y;
```

其具体运算过程如下：

```
   0000000000001001(x)
^
   0000000000000100(y)
————————————————————————
   0000000000001101(x)
^
   0000000000000100(y)
————————————————————————
   0000000000001001(y)
^
   0000000000001101(x)
————————————————————————
   0000000000000100(x)
```

【例 12.4】 输入两个数分别赋予变量 a 和 b，计算 a^b 的值。（**实例位置：资源包\TM\sl\12\4**）

```
#include<stdio.h>
main()
{
    unsigned result;                          /*定义无符号数*/
    int a, b;
    printf("please input a:");
    scanf("%d",&a);
    printf("please input b:");
    scanf("%d",&b);
    printf("a=%d,b=%d", a, b);
    result = a^b;                             /*求 a 与 b “异或” 的结果*/
    printf("\na^b=%u\n", result);
}
```

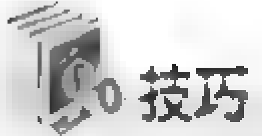
程序运行结果如图 12.4 所示。



图 12.4 a^b

例 12.4 的执行过程如下：

```
  0000000000111000
^ 0000000001001000
------------------
  0000000001110000
```



技巧

“异或”运算经常被用到一些比较简单的加密算法中。

12.2.5 “左移”运算符

“左移”运算符“<<”是双目运算符，其功能是把“<<”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0。

如 a<<2 即把 a 的各二进位向左移动两位。假设 a=39，那么 a 在内存中的存储情况如图 12.5 所示。

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1

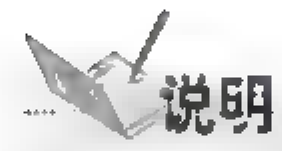
图 12.5　39 在内存中的存储情况

若将 a 左移两位，则在内存中的存储情况如图 12.6 所示。

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0

图 12.6　39 左移两位

a 左移两位后由原来的 39 变成了 156。



说明

实际上左移一位相当于该数乘以 2，将 a 左移两位相当于 a 乘以 4，即 39 乘以 4，但这种情况只限于移出位不含 1 的情况。若是将十进制数 64 左移两位，则移位后的结果将为 0（01000000->00000000），这里因为 64 在左移两位时将 1 移出了（注意这里的 64 是假设以一个字节（即 8 位）存储的）。

【例 12.5】　将 15 先左移两位，将其左移后的结果输出，在这个结果的基础上再左移 3 位，并将结果输出。（**实例位置：资源包\TM\sl\12\5**）

```
#include<stdio.h>
main()
{
    int x=15;
    x=x<<2;                                    /*x 左移两位*/
    printf("the result1 is:%d\n",x);
    x=x<<3;                                    /*x 左移 3 位*/
```

```
    printf("the result2 is:%d\n",x);
}
```

程序运行结果如图 12.7 所示。



图 12.7　左移运算

例 12.5 的执行过程如下：

15 在内存中的存储情况如图 12.8 所示。

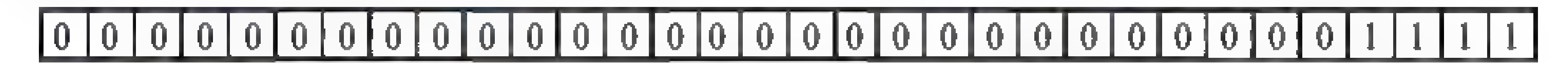


图 12.8　15 在内存中的存储情况

15 左移两位后变为 60，其存储情况如图 12.9 所示。

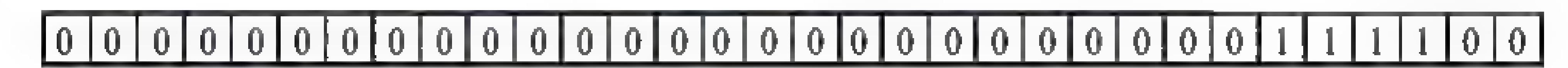


图 12.9　15 左移两位

60 左移 3 位变成 480，其存储情况如图 12.10 所示。

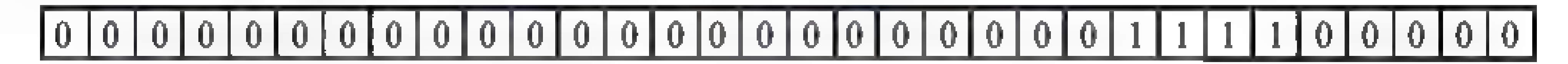
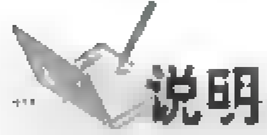


图 12.10　60 左移 3 位

12.2.6　“右移”运算符

右移运算符“>>”是双目运算符，其功能是把“>>”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移动的位数。

例如，a>>2 即把 a 的各二进位向右移动两位，假设 a=00000110，右移两位后为 00000001，a 由原来的 6 变成了 1。



说明

在进行右移时对于有符号数需要注意符号位问题，当为正数时，最高位补 0；而为负数时，最高位是补 0 还是补 1 取决于编译系统的规定。移入 0 的称为“逻辑右移”，移入 1 的称为“算术右移”。

【例 12.6】　将 30 和-30 分别右移 3 位，将所得结果分别输出，在所得结果的基础上再分别右移两位，并将结果输出。（**实例位置：资源包\TM\sl\12\6**）

```
#include<stdio.h>
main()
{
    int x=30,y=-30;
    x=x>>3;                                   /*x 右移 3 位*/
```

```
    y=y>>3;                                          /*y 右移 3 位*/
    printf("the result1 is:%d,%d\n",x,y);
    x=x>>2;                                          /*x 右移两位*/
    y=y>>2;                                          /*x 右移两位*/
    printf("the result2 is:%d,%d\n",x,y);
}
```

程序运行结果如图 12.11 所示。

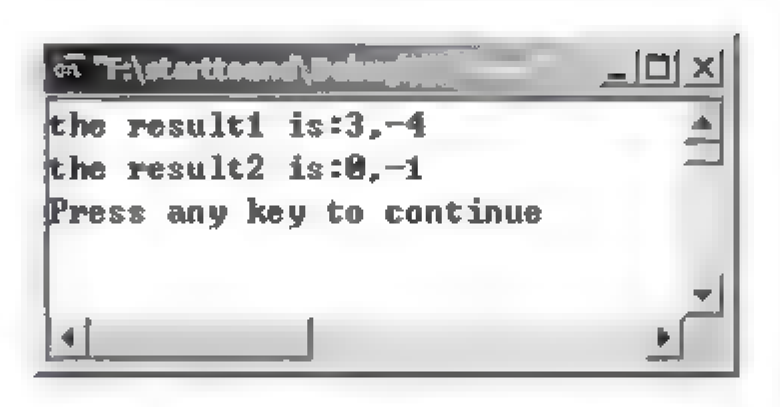


图 12.11　右移运算

例 12.6 的执行过程如下：

30 在内存中的存储情况如图 12.12 所示。

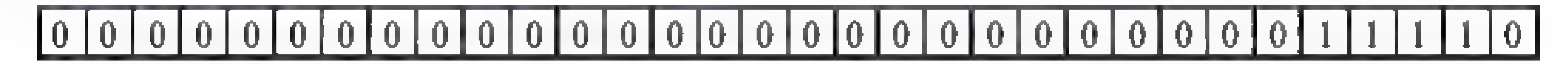


图 12.12　30 在内存中的存储情况

-30 在内存中的存储情况如图 12.13 所示。

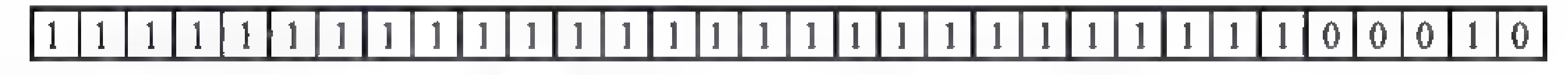


图 12.13　-30 在内存中的存储情况

30 右移 3 位变成 3，其存储情况如图 12.14 所示。

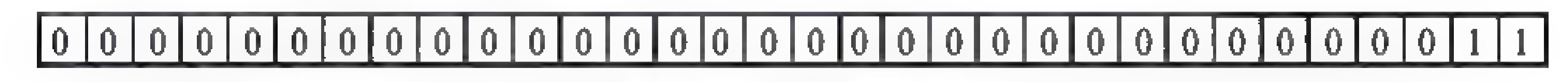


图 12.14　30 右移 3 位

-30 右移 3 位变成-4，其存储情况如图 12.15 所示。

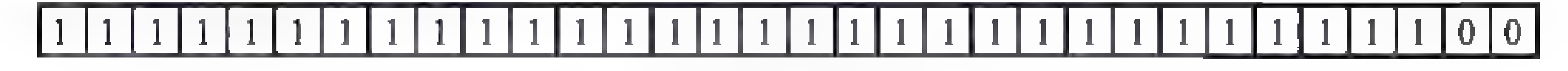


图 12.15　-30 右移 3 位

3 右移两位变成 0，而-4 右移两位则变成-1，如图 12.16 所示。

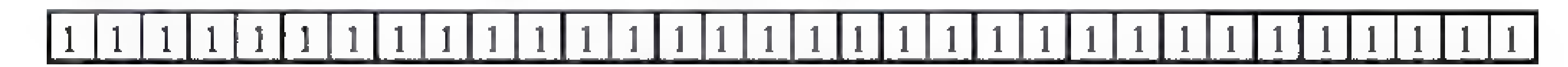


图 12.16　-4 右移两位

从上面的过程中可以发现在 Visual C++ 6.0 中进行的负数右移实质上就是算术右移。



12.3　循环移位

前面讲过了向左移位和向右移位，这里将介绍循环移位的相关内容。什么是循环移位呢？循环移

位就是将移出的低位放到该数的高位，或者将移出的高位放到该数的低位。那么该如何实现这个过程呢？这里先介绍如何实现循环左移。

循环左移的过程如图 12.17 所示。

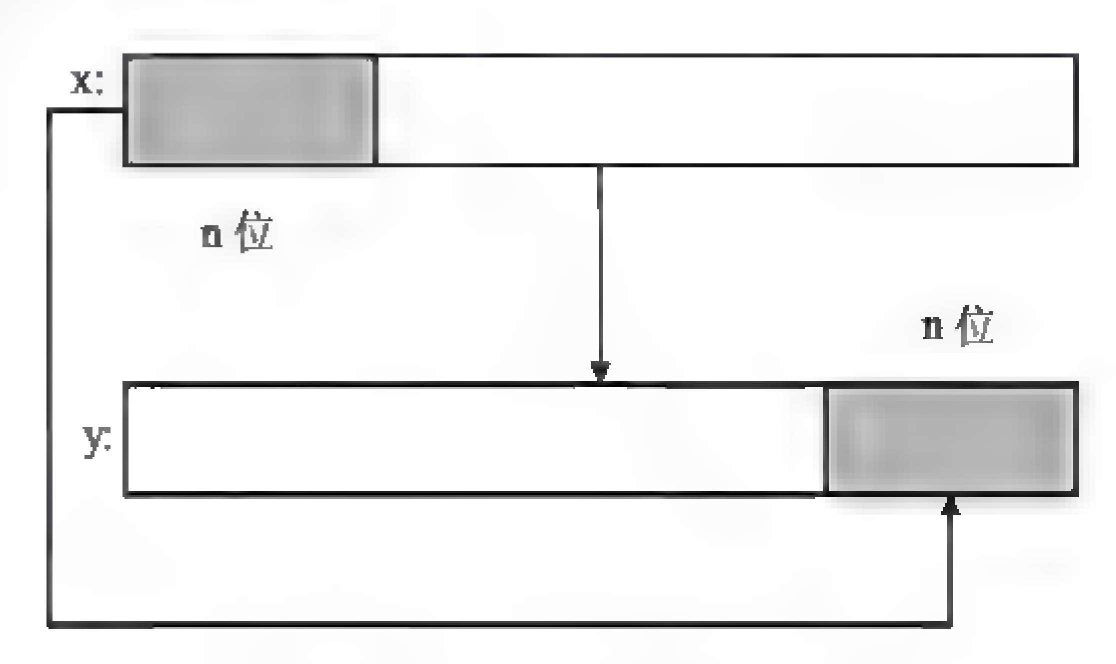


图 12.17　循环左移的过程

实现循环左移的过程如下：

如图 12.17 所示将 x 的左端 n 位先放到 z 中的低 n 位中。由以下语句实现：

```
z=x>>(32-n);
```

将 x 左移 n 位，其右面低 n 位补 0。由以下语句实现：

```
y=x<<n;
```

将 y 与 z 进行按位“或”运算。由以下语句实现：

```
y=y|z;
```

【例 12.7】　编程实现循环左移，具体要求：首先从键盘中输入一个八进制数，然后输入要移位的位数，最后将移位的结果显示在屏幕上。（**实例位置：资源包\TM\sl\12\7**）

```
#include <stdio.h>
left(unsigned value, int n)                                /*自定义左移函数*/
{
    unsigned z;
    z = (value >> (32-n)) | (value << n);                  /*循环左移的实现过程*/
    return z;
}
main()
{
    unsigned a;
    int n;
    printf("please input a number:\n");
    scanf("%o", &a);                                       /*输入一个八进制数*/
    printf("please input the number of displacement（>0）:\n");
    scanf("%d", &n);                                       /*输入要移位的位数*/
    printf("the result is %o:\n", left(a, n));             /*将左移后的结果输出*/
}
```

程序运行结果如图 12.18 所示。

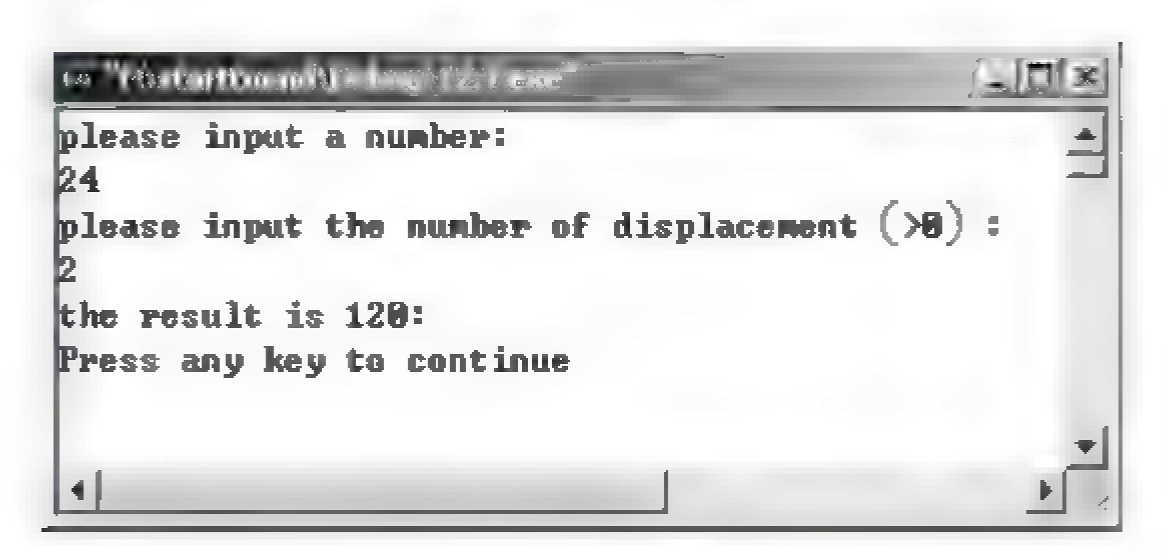


图 12.18　循环左移

循环右移的过程如图 12.19 所示。

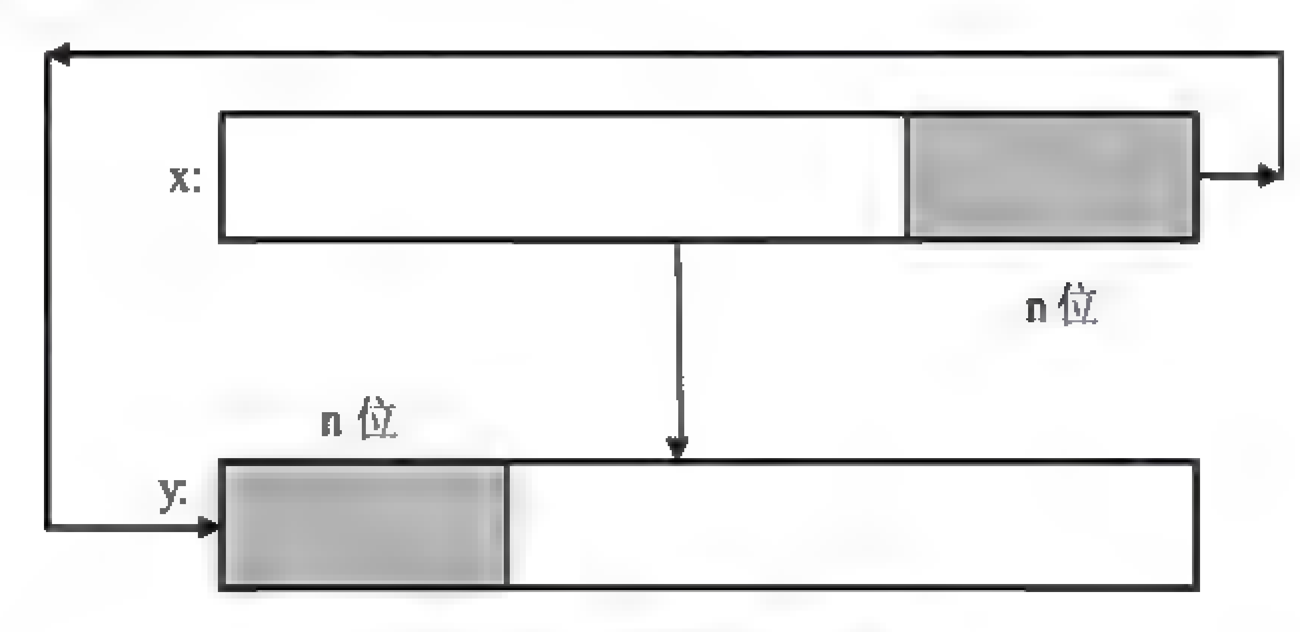


图 12.19　循环右移的过程

如图 12.19 所示，将 x 的右端 n 位先放到 z 中的高 n 位中，由以下语句实现：

```
z=x<<(32-n);
```

将 x 右移 n 位，其左端高 n 位补 0，由以下语句实现：

```
y=x>>n;
```

将 y 与 z 进行按位或运算，由以下语句实现：

```
y=y|z;
```

【例 12.8】　编程实现循环右移，具体要求：首先从键盘中输入一个八进制数，然后输入要移位的位数，最后将移位的结果显示在屏幕上。（**实例位置：资源包\TM\sl\12\8**）

```
#include <stdio.h>
right(unsigned value, int n)                                    /*自定义右移函数*/
{
    unsigned z;
    z = (value << (32-n)) | (value >> n);                       /*循环右移的实现过程*/
    return z;
}
main()
{
    unsigned a;
    int n;
    printf("please input a number:\n");
```

```
    scanf("%o", &a);                                          /*输入一个八进制数*/
    printf("please input the number of displacement（>0）:\n");
    scanf("%d", &n);                                          /*输入要移位的位数*/
    printf("the result is %o:\n", right(a, n));               /*将右移后的结果输出*/
}
```

程序运行结果如图 12.20 所示。

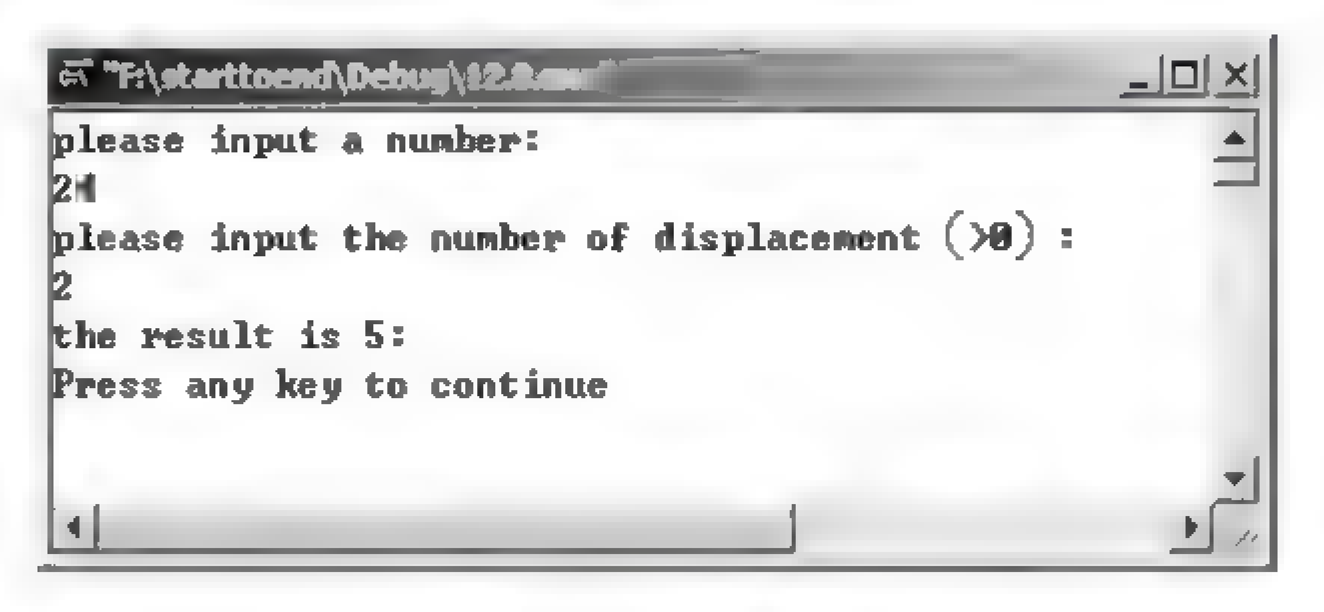


图 12.20　循环右移

12.4　位　　段



12.4.1　位段的概念与定义

位段类型是一种特殊的结构类型，其所有成员的长度均是以二进制位为单位定义的，结构中的成员被称为位段。位段定义的一般形式如下：

```
结构 结构名
{
  类型   变量名 1:长度;
  类型   变量名 2:长度;
  …
  类型   变量名 n:长度;
}
```

一个位段必须被说明是 int、unsigned 或 signed 中的一种。

例如，CPU 的状态寄存器按位段类型定义如下：

```
struct status
   {
     unsigned sign:1;                     /*符号标志*/
     unsigned zero:1;                     /*零标志*/
     unsigned carry:1;                    /*进位标志*/
     unsigned parity:1;                   /*奇偶溢出标志*/
     unsigned half_carry:1;               /*半进位标志*/
     unsigned negative:1;                 /*减标志*/
  } flags;
```

显然，对 CPU 的状态寄存器而言，使用位段类型仅需 1 个字节即可。

又如：

```
struct packed_data
{
unsigned a:2;
unsigned b:1;
unsigned c:1;
unsigned d:2;
}data;
```

可以发现，这里 a、b、c、d 分别占 2 位、1 位、1 位、2 位，如图 12.21 所示。

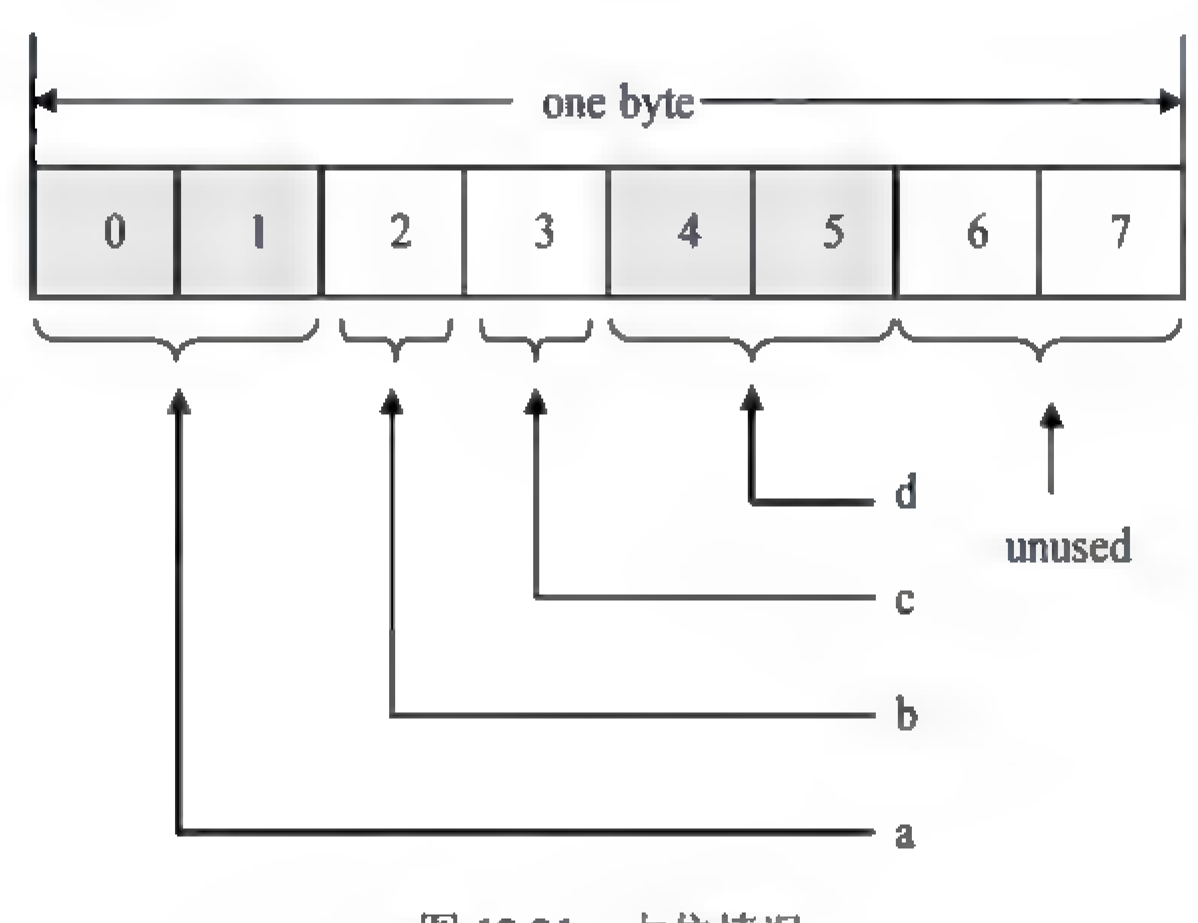


图 12.21　占位情况

12.4.2　位段相关说明

前面介绍了什么是位段，这里针对位段有以下几点加以说明。

☑ 因为位段类型是一种结构类型，所以位段类型和位段变量的定义，以及对位段（即位段类型中的成员）的引用均与结构类型和结构变量相同。

☑ 定义一个如下的位段结构：

```
struct attribute
{
	unsigned font:1;
	unsigned color:1;
	unsigned size:1;
	unsigned dir:1;
};
```

上面定义的位段结构中，各个位段都只占用一个二进制位，如果某个位段需要表示多于两种的状态，也可将该位段设置为占用多个二进制位。如果字体大小有 4 种状态，则可将上面的位段结构改写成如下形式：

```
struct attribute
{
        unsigned font:1;
        unsigned color:1;
        unsigned size:2;
        unsigned dir:1;
};
```

☑ 某一位段要从另一个字节开始存放，可写成如下形式：

```
struct status
  {
        unsigned a:1;
        unsigned b:1;
        unsigned c:1;
        unsigned :0;
        unsigned d:1;
        unsigned e:1;
        unsigned f:1
  }flags;
```

原本 a、b、c、d、e、f 这 6 个位段是连续存储在一个字节中的。由于加入了一个长度为 0 的无名位段，因此其后的 3 个位段从下一个字节开始存储，一共占用两个字节。

☑ 可以使各个位段占满一个字节，也可以不占满一个字节。例如：

```
struct packed_data
{
        unsigned a:2;
        unsigned b:2;
        unsigned c:1;
        int i;
}data;
```

存储形式如图 12.22 所示。

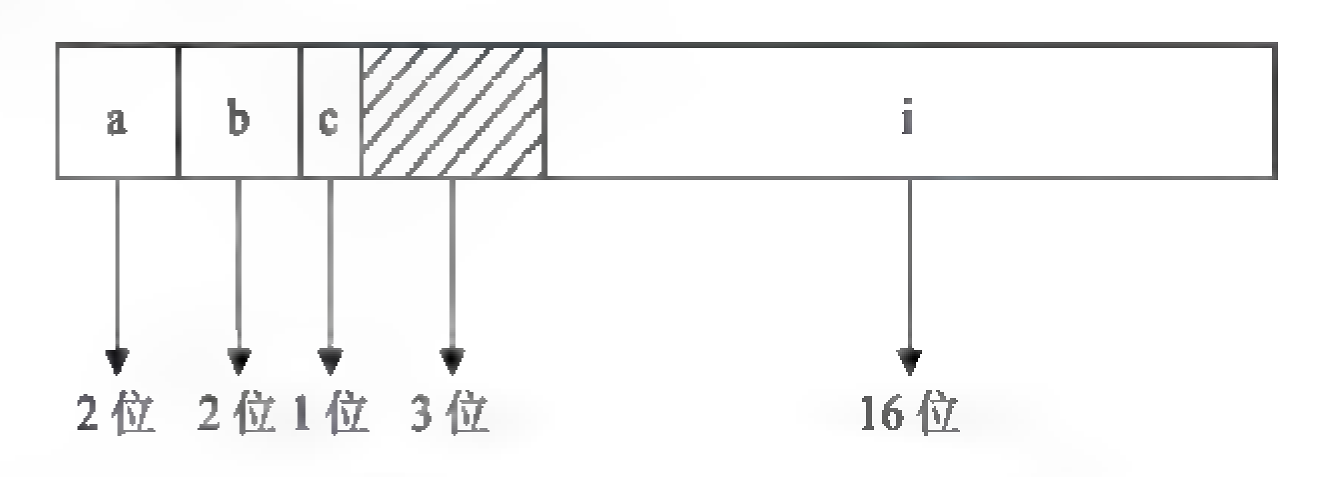


图 12.22 不占满一个字节的情况

☑ 一个位段必须存储在一个存储单元（通常为一字节）中，不能跨两个存储单元。如果本单元不够容纳某位段，则从下一个单元开始存储该位段。

☑ 可以用"%d""%x""%u""%o"等格式字符，以整数形式输出位段。

☑ 在数值表达式中引用位段时，系统自动将位段转换为整型数。

12.5　小　　结

位运算是C语言的一种特殊运算功能，它是以二进制位为单位进行运算的。本章主要介绍了与（&）、或（|）、取反（~）、异或（^）、左移（<<）、右移（>>）6种位运算符，利用位运算可以完成汇编语言的某些功能，如置位、位清零、移位等。

位域在本质上也是结构类型，不过它的成员按二进制位分配内存，其定义、说明及使用的方式都与结构相同。位域可以实现数据的压缩，节省了存储空间的同时也提高了程序的效率。

12.6　实践与练习

1. 任意输入两个数，求这两个数进行“与”和“或”运算之后的结果。**（答案位置：资源包\TM\sl\12\9）**

2. 任意输入一个数，分别求该数“左移”和“右移”运算操作后的结果。**（答案位置：资源包\TM\sl\12\10）**

3. 任意输入一个数，分别对该数进行“循环左移”和“循环右移”操作，并将结果输出。**（答案位置：资源包\TM\sl\12\11）**

第13章

预处理

（视频讲解：40分钟）

预处理是C语言特有的功能，可以使用预处理和具有预处理的功能是C语言与其他高级语言的区别之一。预处理程序包含许多有用的功能，如宏定义、条件编译等，使用预处理功能便于程序的修改、阅读、移植和调试，也便于实现模块化程序设计。

通过阅读本章，您可以：

- 掌握宏定义相关内容
- 掌握文件包含相关内容
- 掌握条件编译相关内容



13.1 宏 定 义

在前面的学习中经常遇到用#define 命令定义符号常量的情况。其实，使用#define 命令就是要定义一个可替换的宏。宏定义是预处理命令的一种，它提供了一种可以替换源代码中字符串的机制。根据宏定义中是否有参数，可以将宏定义分为不带参数的宏定义和带参数的宏定义两种，下面分别进行介绍。

13.1.1 不带参数的宏定义

宏定义指令#define 用来定义一个标识符和一个字符串，以这个标识符来代表这个字符串，在程序中每次遇到该标识符时就用所定义的字符串替换它。宏定义的作用相当于给指定的字符串起一个别名。

不带参数的宏定义一般形式如下：

```
#define  宏名  字符串
```

☑ #表示这是一条预处理命令。
☑ 宏名是一个标识符，必须符合 C 语言标识符的规定。
☑ 字符串可以是常数、表达式、格式字符串等。

例如：

```
#define PI 3.14159
```

该语句的作用是在该程序中用 PI 替代 3.14159，在编译预处理时，每当在源程序中遇到 PI 就自动用 3.14159 代替。

使用#define 进行宏定义的好处是需要改变一个常量时只需改变#define 命令行，整个程序的常量都会改变，大大提高了程序的灵活性。

宏名要简单且意义明确，一般习惯用大写字母表示，以便与变量名相区别。



宏定义不是 C 语句，不需要在行末加分号。

宏名定义后，即可成为其他宏名定义中的一部分。例如，下面代码定义了正方形的边长 SIDE、周长 PERIMETER 及面积 AREA 的值。

```
#define  SIDE  5
#define  PERIMETER  4*SIDE
#define  AREA  SIDE*SIDE
```

前面强调过，宏替换是以串代替标识符。因此，如果希望定义一个标准的邀请语，可编写如下代码：

```
#define  STANDARD  "You are welcome to join us."
printf(STANDARD);
```

编译程序遇到标识符 STANDARD 时，就用"You are welcome to join us."替换。

对于编译程序，printf 语句如下形式是等效的：

```
printf("possible use of 'l' before definition in function main");
```

关于不带参数的宏定义，有以下几点需要强调。

☑　如果在串中含有宏名，则不进行替换。例如：

```
#include<stdio.h>
#define TEST "this is an example"
main()
{
    char exp[30]="This TEST is not that TEST";                /*定义字符数组并赋初值*/
    printf("%s\n",exp);
}
```

该段代码输入结果如图 13.1 所示。

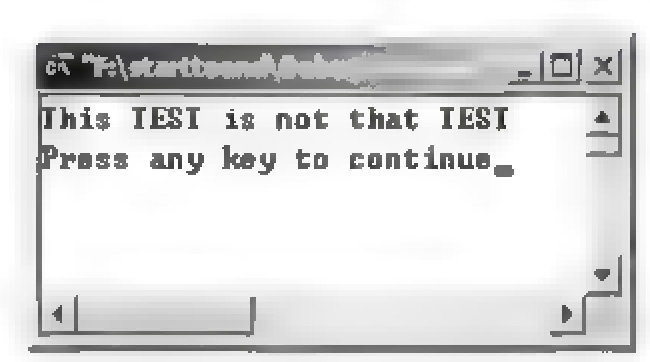


图 13.1　在串中含有宏名

注意，上面程序字符串中的 TEST 并没有用"this is an example"来替换，因此如果串中含有宏名，则不进行替换。

☑　如果串长于一行，可以在该行末尾用反斜杠“\”续行。

☑　#define 命令出现在程序中函数的外面，宏名的有效范围为定义命令之后到此源文件结束。

注意

在编写程序时通常将所有的#define 放到文件的开始处或独立的文件中，而不是将它们分散到整个程序中。

☑　可以用#undef 命令终止宏定义的作用域，例如：

```
#include<stdio.h>
#define TEST "this is an example"
main()
{
    printf(TEST);
    #undef TEST
}
```

☑　宏定义用于预处理命令，它不同于定义的变量，只做字符替换，不分配内存空间。

13.1.2　带参数的宏定义

带参数的宏定义不是简单的字符串替换，还要进行参数替换。其一般形式如下：

```
#define 宏名(参数表)字符串
```

【例 13.1】 对两个数实现乘法加法混合运算。（**实例位置：资源包\TM\sl\13\1**）

```
#include<stdio.h>
#define MIX(a,b) ((a)*(b)+(b))                        /*宏定义求两个数的混合运算*/
main()
{
    int x=5,y=9;
    printf("x,y:\n");
    printf("%d,%d\n",x,y);
    printf("the min number is:%d\n",MIX(x,y));          /*宏定义调用*/
}
```

程序运行结果如图 13.2 所示。

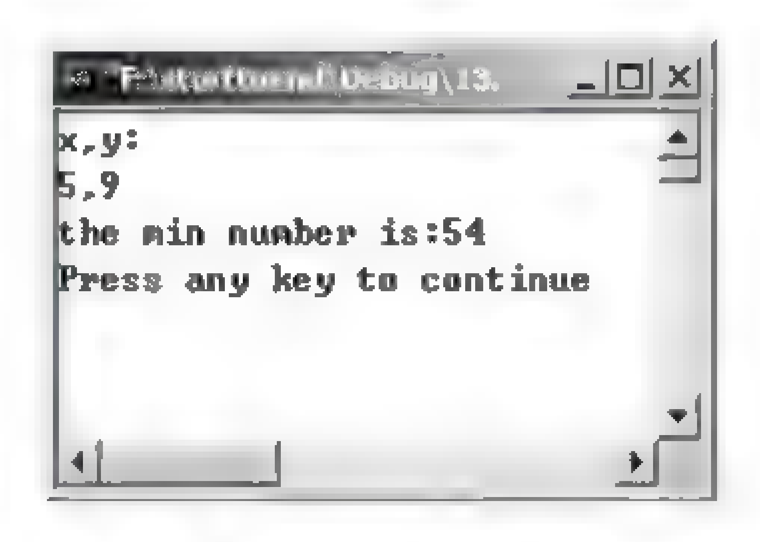


图 13.2 混合运算

当编译该程序时，由 MIX(a,b)定义的表达式被替换，x 和 y 用作操作数，即 printf 语句被替换后取如下形式：

```
printf("the min number is: %d",((a)*(b)+(b)));
```

用宏替换代替实在的函数的一个好处是可以提升代码的速度，因为不再存在函数调用。但提升速度也是有代价的：由于重复编码，而增加了程序长度。

对于带参数的宏定义有以下几点需要强调。

☑ 宏定义时，参数要加括号。如不加括号，则结果可能是正确的，也可能是错误的。

如例 13.1，当参数 x=10，y=9 时，在参数不加括号的情况下调用 MIX(x,y)，可以正确地输出结果。当 x=10，y=3+4 时，在参数不加括号的情况下调用 MIX(x,y)，则输出的结果是错误的，因为此时调用的 MIX(x,y)执行情况如下：

```
(10*3+4+3+4);
```

此时计算出的结果是 41，而实际上希望得出的结果是 77。为了避免出现上面这种情况，在进行宏定义时要在参数外面加上括号。

☑ 宏扩展必须使用括号来保护表达式中低优先级的操作符，以确保调用时能达到想要的效果。

如果例 13.1 宏扩展外没有加括号，则调用：

```
5*MIX(x,y)
```

会被扩展为：

```
5*(a)*(b)+(b)
```

而本意是希望得到：

```
5*((a)*(b)+(b))
```

同样，只要在宏扩展时加上括号，就能避免这种错误发生。

- ☑ 对带参数的宏的展开，只是将语句中的宏名后面括号内的实参字符串代替#define 命令行中的形参。
- ☑ 在宏定义时，宏名与带参数的括号之间不可以加空格，否则会将空格以后的字符都作为替代字符串的一部分。
- ☑ 在带参宏定义中，形式参数不分配内存单元，因此不必做类型定义。



13.2　#include 指令

在一个源文件中使用#include 指令可以将另一个源文件的全部内容包含进来，也就是将另外的文件包含到本文件之中。#include 使编译程序将另一源文件嵌入带有#include 的源文件，被读入的源文件必须用双引号或尖括号括起来。例如：

```
#include "stdio.h"
#include <stdio.h>
```

这两行代码均使用 C 编译程序读入并编译，用于处理磁盘文件库的子程序。

上面给出了双引号和尖括号的形式，这两者之间的区别是：用尖括号时，系统到存放 C 库函数头文件所在的目录中寻找要包含的文件，这为标准方式；用双引号时，系统先在用户当前目录中寻找要包含的文件，若找不到，再到存放 C 库函数头文件所在的目录中寻找要包含的文件。通常情况下，如果为调用库函数用#include 命令来包含相关的头文件，则用尖括号可以节省查找的时间。如果要包含的是用户自己编写的文件，一般用双引号。用户自己编写的文件通常是在当前目录中，如果文件不在当前目录中，双引号可给出文件路径。

将文件嵌入#include 命令中的文件内是可行的，这种方式称为嵌套的嵌入文件，嵌套层次依赖于具体实现，如图 13.3 所示。

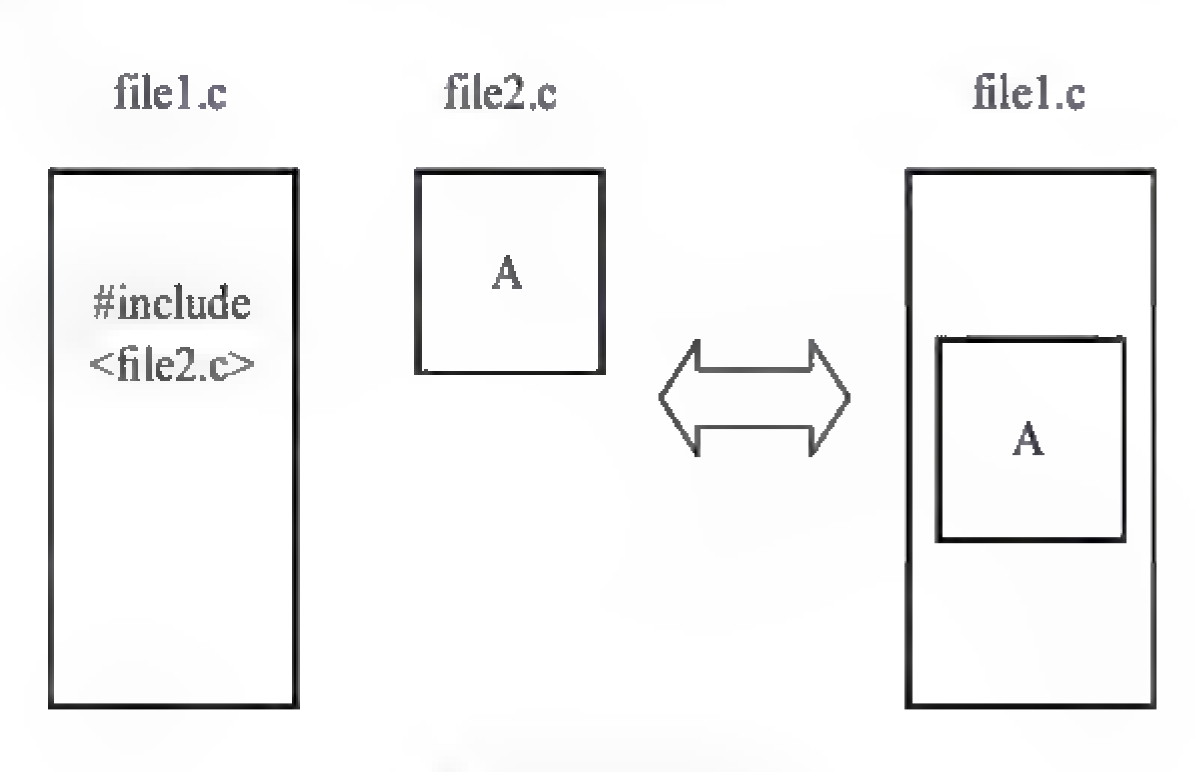


图 13.3　文件包含

【例 13.2】 文件包含应用。（**实例位置：资源包\TM\sl\13\2**）

（1）文件 f1.h

```
#define P printf
#define S scanf
#define D "%d"
#define C "%c"
```

（2）文件 f2.c

```
#include<stdio.h>
#include<f1.h>                              /*包含文件 f1.h*/
main()
{
    int a;
    P("please input:\n");
    S(D,&a);                                /*调用 f1 中的宏定义*/
    P("the number is:\n");
    P(D,a);                                 /*调用 f1 中的宏定义*/
    P("\n");
    P(C,a);
    P("\n");
}
```

程序运行结果如图 13.4 所示。

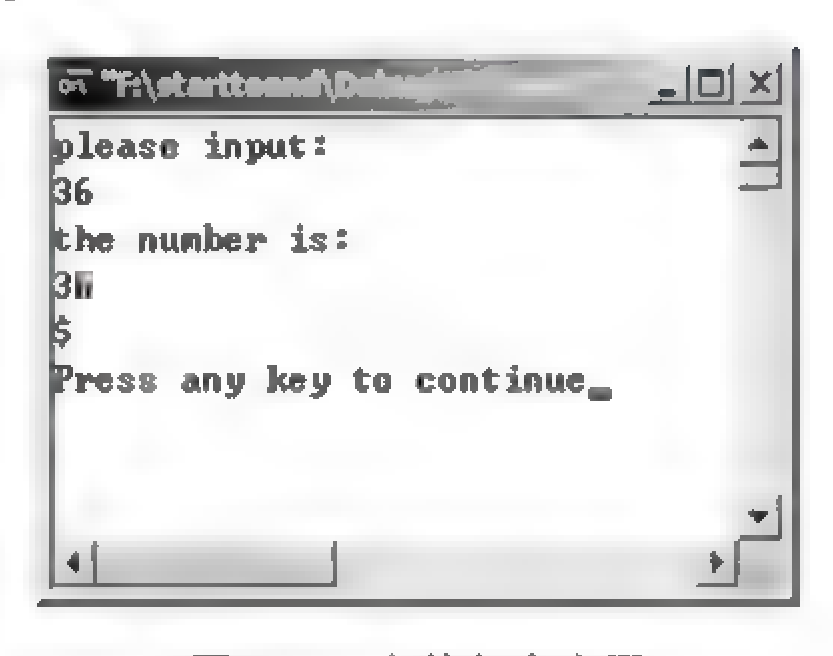


图 13.4 文件包含应用

经常用在文件头部的被包含的文件称为“标题文件”或“头部文件”，一般以.h 为后缀，如本实例中的 f1.h。

一般情况下，将如下内容放到.h 文件中：

☑ 宏定义。
☑ 结构、联合和枚举声明。
☑ typedef 声明。
☑ 外部函数声明。
☑ 全局变量声明。

使用文件包含为实现程序修改提供了方便。当需要修改某些参数时，不必逐个修改程序，只需修改一个文件（头部文件）即可。

关于“文件包含”有以下几点需要注意。

☑ 一个#include 命令只能指定一个被包含的文件。

☑ 文件包含是可以嵌套的，即在一个被包含文件中还可以包含另一个被包含文件。

☑ 若 file1.c 中包含文件 file2.h，那么在预编译后就成为一个文件，而不是两个文件。这时如果 file2.h 中有全局静态变量，则该全局变量在 file1.c 文件中也有效，这时不需要再用 extern 声明。



13.3 条件编译

预处理器提供了条件编译功能，一般情况下，源程序中所有的行都参加编译。如果只希望其中一部分内容在满足一定条件时才进行编译，这时就需要使用一些条件编译命令。使用条件编译可以非常方便地处理程序的调试版本和正式版本，同时还会增强程序的可移植性。

13.3.1 #if 命令

#if 的基本含义是：如果#if 命令后的参数表达式为真，则编译#if 到#endif 之间的程序段，否则跳过这段程序。#endif 命令用来表示#if 段的结束。

#if 命令的一般形式如下：

```
#if 常数表达式
    语句段
#endif
```

如果常数表达式为真，则该段程序被编译，否则跳过不编译。

【例 13.3】 #if 应用。（**实例位置：资源包\TM\sl\13\3**）

```
#include<stdio.h>
#define NUM 50
main()
{
    int i=0;
#if NUM>50                                  /*判断 NUM 是否大于 50*/
    i++;
#endif
#if NUM==50
    i=i+50;
#endif
#if NUM<50
    i--;
#endif
    printf("Now i is:%d\n",i);
}
```

程序运行结果如图 13.5 所示。



图 13.5　#if 应用

若将语句：

```
#define NUM 50
```

改为：

```
#define NUM 10
```

则程序运行结果如图 13.6 所示。

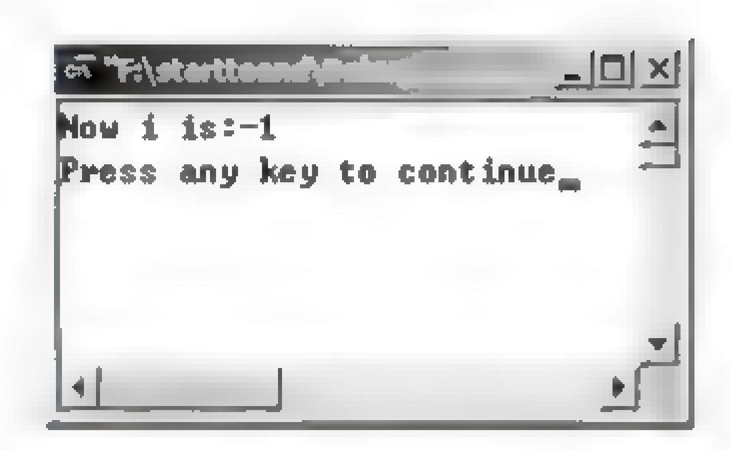


图 13.6　NUM 为 10 时的运行结果

同样，若将语句：

```
#define NUM 50
```

改为：

```
#define NUM 100
```

则程序运行结果如图 13.7 所示。

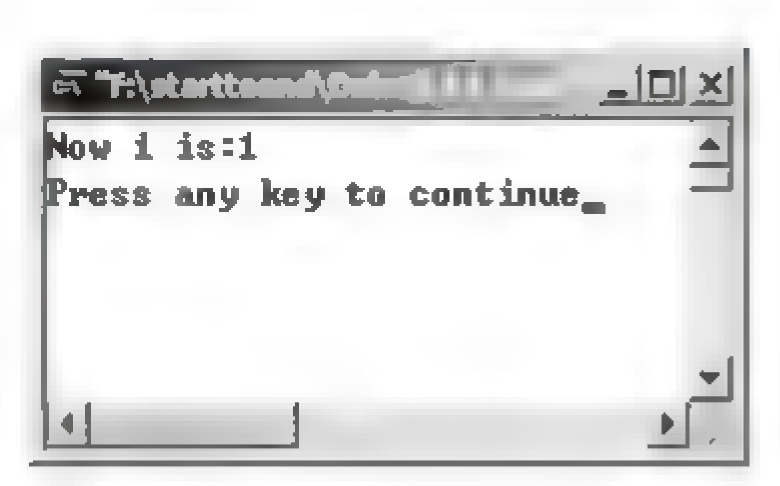


图 13.7　当 NUM 为 100 时的运行结果

#else 的作用是为#if 为假时提供另一种选择，其作用和前面讲过的条件判断中的 else 相近。

【例 13.4】　#else 应用。（**实例位置：资源包\TM\sl\13\4**）

```
#include<stdio.h>
#define NUM 50
main()
{
```

```
    int i=0;
#if NUM>50
    i++;
#else
#if NUM<50
    i--;
#else
    i=i+50;
#endif
#endif
    printf("i is:%d\n",i);
}
```

程序运行结果如图 13.8 所示。

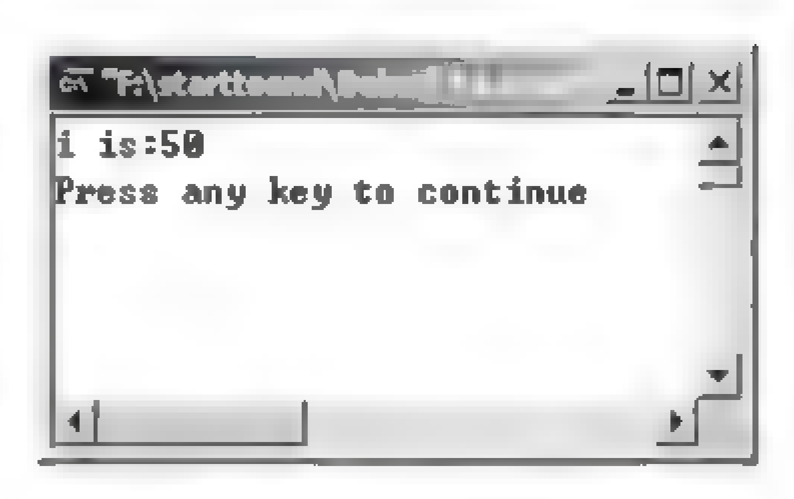


图 13.8　#else 应用

#elif 指令用来建立一种"如果……或者如果……"这样阶梯状多重编译操作选择，这与多分支 if 语句中的 else if 类似。

#elif 的一般形式如下：

```
#if 表达式
语句段
#elif 表达式 1
语句段
#elif 表达式 2
语句段
…
#elif 表达式 n
语句段
#endif
```

在运行结果不发生改变的前提下可将例 13.4 改写成如下形式。

【例 13.5】　#elif 应用。(**实例位置：资源包\TM\sl\13\5**)

```
#include<stdio.h>
#define NUM 50
main()
{
    int i=0;
   #if NUM>50
       i++;
```

```
    #elif NUM==50
        i=i+50;
    #else
        i--;
    #endif
     printf("i is:%d\n",i);
}
```

13.3.2 #ifdef 及#ifndef 命令

在#if 条件编译命令中，需要判断符号常量所定义的具体值。但有时并不需要判断具体值，只需要知道这个符号常量是否被定义了，这时就不需要使用#if，而可以采用另一种条件编译的方法，即#ifdef 与#ifndef 命令，分别表示“如果有定义”及“如果无定义”。下面就对这两个命令进行介绍。

#ifdef 的一般形式如下：

```
#ifdef 宏替换名
语句段
#endif
```

其含义是：如果宏替换名已被定义过，则对“语句段”进行编译；如果未定义#ifdef 后面的宏替换名，则不对语句段进行编译。

#ifdef 可与#else 连用，形式如下：

```
#ifdef 宏替换名
语句段 1
#else
语句段 2
#endif
```

其含义是：如果宏替换名已被定义过，则对“语句段 1”进行编译；如果未定义#ifdef 后面的宏替换名，则对“语句段 2”进行编译。

#ifndef 的一般形式如下：

```
#ifndef 宏替换名
语句段
#endif
```

其含义是：如果未定义#ifndef 后面的宏替换名，则对“语句段”进行编译；如果定义#ifndef 后面的宏替换名，则不执行语句段。

同样，#ifndef 也可以与#else 连用，构成的一般形式如下：

```
#ifndef 宏替换名
语句段 1
#else
语句段 2
#endif
```

其含义是：如果未定义#ifndef 后面的宏替换名，则对“语句段 1”进行编译；如果定义#ifndef 后面的宏替换名，则对“语句段 2”进行编译。

【例 13.6】　#ifdef 和#ifndef 的具体应用。（**实例位置：资源包\TM\sl\13\6**）

```
#include<stdio.h>
#define STR "diligence is the parent of success\n"
main()
{
    #ifdef STR
        printf(STR);
    #else
        printf("idleness is the root of all evil\n");
    #endif
    printf("\n");
    #ifndef ABC
        printf("idleness is the root of all evil\n");
    #else
        printf(STR);
    #endif
}
```

程序运行结果如图 13.9 所示。

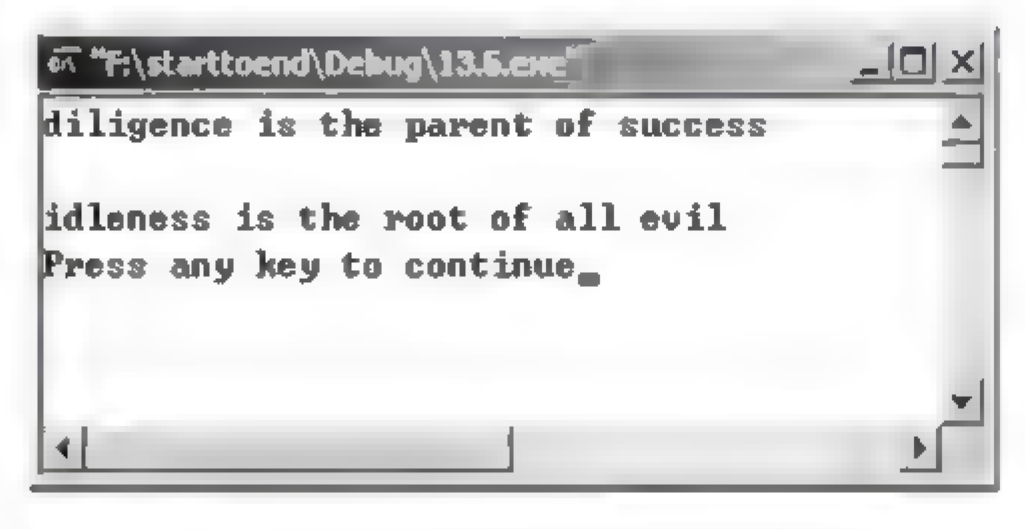


图 13.9　#ifdef 和#ifndef 的具体应用

13.3.3　#undef 命令

前文介绍#define 命令时提到过#undef 命令，使用#undef 命令可以删除事先定义好的宏定义。

#undef 命令的一般形式如下：

```
#undef 宏替换名
```

例如：

```
#define MAX_SIZE 100
char array[MAX_SIZE];
#undef   MAX_SIZE
```

在上述代码中，首先使用#define 定义标识符 MAX_SIZE，然后使用#undef 删除宏定义。也就是说，直到遇到#undef 语句之前，MAX_SIZE 的定义都是有效的。



说明

#undef 的主要目的是将宏名局限在仅需要它们的代码段中。

13.3.4 #line 命令

#line 命令用于显示_LINE_与_FILE_的内容。_LINE_存放当前编译行的行号，_FILE_存放当前编译的文件名。

#line 命令的一般形式如下：

```
#line 行号["文件名"]
```

其中，行号为任一正整数，可选的文件名为任意有效文件标识符。行号为源程序中当前行号，文件名为源文件的名字。#line 命令主要用于调试及其他特殊应用。

【例 13.7】 输出行号。（**实例位置：资源包\TM\sl\13\7**）

```
#line 100 "13.7.C"
#include<stdio.h>
main()
{
printf("1.当前行号：%d\n",__LINE__);
printf("2.当前行号：%d\n",__LINE__);
}
```

程序运行结果如图 13.10 所示。



图 13.10 输出行号

13.3.5 #pragma 命令

1. #pragma 命令

#pragma 命令的作用是设定编译器的状态，或者指示编译器完成一些特定的动作。

#pragma 命令的一般形式如下：

```
#pragma 参数
```

参数可分为以下几种：

☑ message 参数：在编译信息窗口中输出相应的信息。

- ☑ code_seg 参数：设置程序中函数代码存放的代码段。
- ☑ once 参数：保证头文件被编译一次。

2. 预定义宏名

ANSI 标准说明了以下 5 个预定义宏替换名。

- ☑ __LINE__：当前被编译代码的行号。
- ☑ __FILE__：当前源程序的文件名称。
- ☑ __DATE__：当前源程序的创建日期。
- ☑ __TIME__：当前源程序的创建时间。
- ☑ __STDC__：用来判断当前编译器是否为标准 C。若其值为 1，则表示符合标准 C，否则不是标准 C。

如果编译不是标准的，则可能仅支持以上宏名中的几个，或根本不支持。编译程序有时还提供其他预定义的宏名。

注意

宏名的书写比较特别，书写时两边都要由下画线构成。

13.4 小　　结

本章主要讲解了宏定义、文件包含、条件编译这 3 方面内容。宏定义是用一个标识符来表示一个字符串，在宏调用中将用该字符串代换宏名。宏定义分为带参数和不带参数两种形式。文件包含是预处理的一个重要功能，可用于将多个源文件连接成一个源文件进行编译，并生成一个目标文件。条件编译允许只编译源程序中满足条件的程序段，从而减少内存开销，并提高了程序的效率。

13.5 实践与练习

1. 输入两个整数，求它们的乘积，用带参的宏实现。（**答案位置：资源包\TM\sl\13\8**）
2. 分别用函数和带参的宏，从 3 个数中找出最小数。（**答案位置：资源包\TM\sl\13\9**）

第14章

文件

（视频讲解：58分钟）

文件是程序设计中的一个重要概念。在现代计算机的应用领域中，数据处理是一个重要方面，要实现数据处理往往是要通过文件的形式来完成的。本章就来介绍如何将数据写入文件和从文件中读取数据。

通过阅读本章，您可以：

- 了解文件的概念
- 掌握文件的基本操作
- 掌握文件的不同读写方法
- 掌握文件的定位


视频讲解

14.1　文 件 概 述

“文件”是指一组相关数据的有序集合。这个数据集有一个名称，叫作文件名。通常情况下，使用计算机也就是在使用文件。在前面的程序设计中介绍了输入和输出，即从标准输入设备（键盘）输入，由标准输出设备（显示器或打印机）输出。不仅如此，我们也常把磁盘作为信息载体，用于保存中间结果或最终数据。在使用一些字处理工具时，会打开一个文件将磁盘的信息输入内存，通过关闭一个文件来实现将内存数据输出到磁盘。这时的输入和输出是针对文件系统的，因此文件系统也是输入和输出的对象。

所有文件都可以通过流进行输入、输出操作。与文本流和二进制流相对应，文件可以分为文本文件和二进制文件两大类。

- ☑ 文本文件，也称为 ASCII 文件。这种文件在保存时，每个字符对应一个字节，用于存放对应的 ASCII 码。
- ☑ 二进制文件，不是保存 ASCII 码，而是按二进制的编码方式来保存文件内容。

文件可以从不同的角度进行具体的分类：

- ☑ 从用户的角度（或所依附的介质）看，文件可分为普通文件和设备文件两种。
  - ➢ 普通文件是指驻留在磁盘或其他外部介质上的一个有序数据集。
  - ➢ 设备文件是指与主机相连的各种外部设备，如显示器、打印机、键盘等。在操作系统中，把外部设备也看作一个文件来进行管理，把它们的输入、输出等同于对磁盘文件的读和写。
- ☑ 按文件内容看，可分为源文件、目标文件、可执行文件、头文件和数据文件等。

在 C 语言中，文件操作都是由库函数来完成的。本章将介绍主要的文件操作函数。


视频讲解

14.2　文件基本操作

文件的基本操作包括文件的打开和关闭。除了标准的输入、输出文件外，其他所有的文件都必须先打开再使用，使用后还必须关闭该文件。

14.2.1　文件指针

文件指针是一个指向文件有关信息的指针，这些信息包括文件名、状态和当前位置，它们保存在一个结构体变量中。在使用文件时需要在内存中为其分配空间，用来存放文件的基本信息。该结构体类型是由系统定义的，C 语言规定该类型为 FILE 型，其声明如下：

```
typedef struct
{
    short level;
    unsigned flags;
```

```
    char fd;
    unsigned char hold;
    short bsize;
    unsigned char *buffer;
    unsigned ar *curp;
    unsigned istemp;
    short token;
}FILE;
```

从上面的结构可以发现，使用 typedef 定义了一个 FILE 为该结构体类型。在编写程序时可直接使用上面定义的 FILE 类型来定义变量，注意，在定义变量时不必将结构体内容全部给出，只需写成如下形式：

```
FILE *fp;
```



说明

fp 是一个指向 FILE 类型的指针变量。

14.2.2 文件的打开

fopen 函数用来打开一个文件，打开文件的操作就是创建一个流。fopen 函数的原型在 stdio.h 中，其调用的一般形式如下：

```
FILE *fp;
fp=fopen(文件名,使用文件方式);
```

其中，“文件名”是将要被打开文件的文件名，“使用文件方式”是指对打开的文件要进行读还是写。使用文件方式如表 14.1 所示。

表 14.1 使用文件方式

文件使用方式	含 义
r（只读）	打开一个文本文件，只允许读数据
w（只写）	打开或建立一个文本文件，只允许写数据
a（追加）	打开一个文本文件，并在文件末尾写数据
rb（只读）	打开一个二进制文件，只允许读数据
wb（只写）	打开或建立一个二进制文件，只允许写数据
ab（追加）	打开一个二进制文件，并在文件末尾写数据
r+（读写）	打开一个文本文件，允许读和写
w+（读写）	打开或建立一个文本文件，允许读写
a+（读写）	打开一个文本文件，允许读，或在文件末追加数据
rb+（读写）	打开一个二进制文件，允许读和写
wb+（读写）	打开或建立一个二进制文件，允许读和写
ab+（读写）	打开一个二进制文件，允许读，或在文件末追加数据

如果要以只读方式打开文件名为123的文本文档文件，应写成如下形式：

```
FILE *fp;
fp=("123.txt","r");
```

如果使用fopen函数打开文件成功，则返回一个有确定指向的FILE类型指针；若打开失败，则返回NULL。通常打开失败的原因有以下3个方面：

- ☑ 指定的盘符或路径不存在。
- ☑ 文件名中含有无效字符。
- ☑ 以r模式打开一个不存在的文件。

14.2.3 文件的关闭

文件在使用完毕后，应使用fclose函数将其关闭。fclose函数和fopen函数一样，原型也在stdio.h中，调用的一般形式如下：

```
fclose(文件指针);
```

例如：

```
fclose(fp);
```

fclose函数也带回一个值，当正常完成关闭文件操作时，fclose函数返回值为0，否则返回EOF。



说明

在程序结束之前应关闭所有文件，这样做的目的是防止因为没有关闭文件而造成的数据流失。

14.3 文件的读写



打开文件后，即可对文件进行读出或写入的操作。C语言提供了丰富的文件操作函数，本节将对其进行详细介绍。

14.3.1 fputc 函数

fputc函数的一般形式如下：

```
ch=fputc(ch,fp);
```

该函数的作用是把一个字符写到磁盘文件（fp所指向的是文件）中。其中，ch是要输出的字符，它可以是一个字符常量，也可以是一个字符变量。fp是文件指针变量，如果函数输出成功，则返回值就是输出的字符；如果输出失败，则返回EOF。

【例 14.1】 编程实现向E:\exp01.txt中写入“forever…forever…”，以“#”结束输入。（**实例位**

置：资源包\TM\sl\14\1）

```
#include<stdio.h>
main()
{
    FILE *fp;                                    /*定义一个指向 FILE 类型结构体的指针变量*/
    char ch;                                     /*定义变量为字符型*/
    if((fp = fopen("E:\\exp01.txt", "w")) == NULL)   /*以只写方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(0);
    }
    ch = getchar();                              /*getchar 函数带回一个字符赋予 ch*/
    while(ch != '#')                             /*当输入"#"时结束循环*/
    {
        fputc(ch, fp);                           /*将读入的字符写到磁盘文件中*/
        ch = getchar();                          /*getchar 函数继续带回一个字符赋给 ch*/
    }
    fclose(fp);                                  /*关闭文件*/
}
```

当输入如图 14.1 所示的内容时，则 E:\exp01.txt 文件中的内容如图 14.2 所示。



图 14.1　运行界面



图 14.2　文件中的内容

14.3.2　fgetc 函数

fgetc 函数的一般形式如下：

```
ch=fgetc(fp);
```

该函数的作用是从指定的文件（fp 指向的文件）读入一个字符赋予 ch。需要注意的是，该文件必须以读或读写的方式打开。当函数遇到文件结束符时，将返回一个文件结束标志 EOF。

【例 14.2】　要求在程序执行前创建文件 E:\exp02.txt，文档内容为“even the wise are not always free from error;no man is wise at all times”，在屏幕中显示出该文件内容。（**实例位置：资源包\TM\sl\14\2**）

```
#include<stdio.h>
main()
{
    FILE *fp;                                    /*定义一个指向 FILE 类型结构体的指针变量*/
    char ch;                                     /*定义变量及数组为字符型*/
    fp = fopen("e:\\exp02.txt", "r");            /*以只读方式打开指定文件*/
    ch = fgetc(fp);                              /*fgetc 函数带回一个字符赋予 ch*/
```

```
    while(ch != EOF)                               /*当读入的字符值等于 EOF 时结束循环*/
    {
        putchar(ch);                               /*将读入的字符输出到屏幕上*/
        ch = fgetc(fp);                            /*fgetc 函数继续带回一个字符赋予 ch*/
    }
    fclose(fp);                                    /*关闭文件*/
}
```

运行程序，显示效果如图 14.3 所示。



图 14.3　读取磁盘文件

14.3.3　fputs 函数

fputs 函数与 fputc 函数类似，区别在于 fputc 函数每次只向文件中写入一个字符，而 fputs 函数每次向文件中写入一个字符串。

fputs 函数的一般形式如下：

```
fputs(字符串,文件指针)
```

该函数的作用是向指定的文件写入一个字符串，其中字符串可以是字符串常量，也可以是字符数组名、指针或变量。

【例 14.3】　向指定的磁盘文件中写入字符串“gone with the wind”。（**实例位置：资源包\TM\sl\14\3**）

```
#include<stdio.h>
#include<process.h>
main()
{
    FILE *fp;
    char filename[30],str[30];                     /*定义两个字符型数组*/
    printf("please input filename:\n");
    scanf("%s",filename);                          /*输入文件名*/
    if((fp=fopen(filename,"w"))==NULL)             /*判断文件是否打开失败*/
    {
        printf("can not open!\npress any key to continue:\n");
        getchar();
        exit(0);
    }
    printf("please input string:\n");              /*提示输入字符串*/
    getchar();
    gets(str);
    fputs(str,fp);                                 /*将字符串写入 fp 所指向的文件中*/
```

```
    fclose(fp);
}
```

程序运行界面如图 14.4 所示。

如图 14.5 所示为写入文件中的内容。

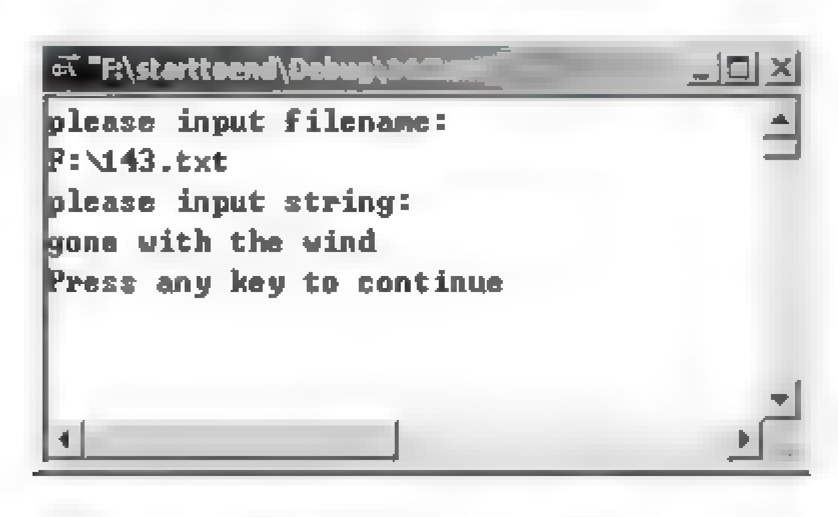


图 14.4　运行界面



图 14.5　写入文件中的内容

14.3.4　fgets 函数

fgets 函数与 fgetc 函数类似，区别在于 fgetc 函数每次从文件中读出一个字符，而 fgets 函数每次从文件中读出一个字符串。

fgets 函数的一般形式如下：

```
fgets(字符数组名,n,文件指针);
```

该函数的作用是从指定的文件中读一个字符串到字符数组中。n 表示所得到的字符串中字符的个数（包含“\0”）。

【例 14.4】　读取任意磁盘文件中的内容。（**实例位置：资源包\TM\sl\14\4**）

```
#include<stdio.h>
#include<process.h>
main()
{
    FILE *fp;
    char filename[30],str[30];                    /*定义两个字符型数组*/
    printf("please input filename:\n");
    scanf("%s",filename);                         /*输入文件名*/
    if((fp=fopen(filename,"r"))==NULL)            /*判断文件是否打开失败*/
    {
        printf("can not open!\npress any key to continue\n");
        getchar();
        exit(0);
    }
    fgets(str,sizeof(str),fp);                    /*读取磁盘文件中的内容*/
    printf("%s",str);
    fclose(fp);
}
```

所要读取的磁盘文件中的内容如图 14.6 所示。

程序运行界面如图 14.7 所示。



图 14.6　文件中的内容

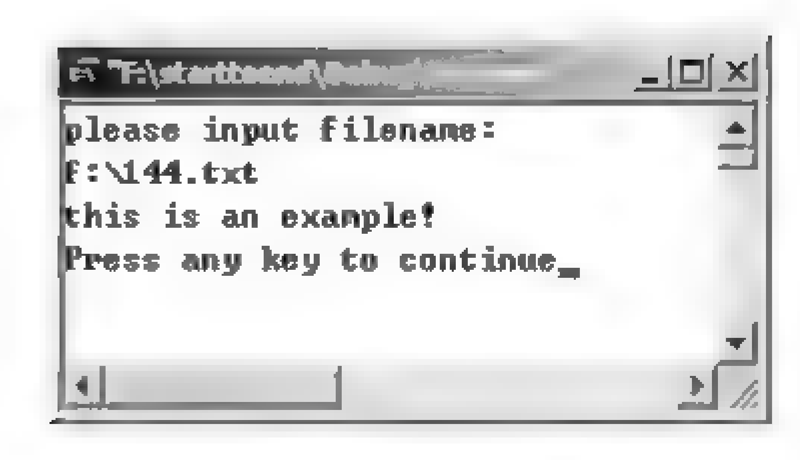


图 14.7　运行界面

14.3.5　fprintf 函数

前面讲过 printf 和 scanf 函数，两者都是格式化读写函数。下面要介绍的 fprintf 和 fscanf 函数，与 printf 和 scanf 函数的作用相似。它们最大的区别就是读写的对象不同，fprintf 和 fscanf 函数读写的对象不是终端，而是磁盘文件。

fprintf 函数的一般形式如下：

```
ch=fprintf(文件类型指针,格式字符串,输出列表);
```

例如：

```
fprintf(fp,"%d",i);
```

它的作用是将整型变量 i 的值以“%d”的格式输出到 fp 指向的文件中。

【例 14.5】　将数字 88 以字符的形式写入磁盘文件中。(**实例位置：资源包\TM\sl\14\5**)

```
#include<stdio.h>
#include<process.h>
main()
{
    FILE *fp;
    int i=88;
    char filename[30];                          /*定义一个字符型数组*/
    printf("please input filename:\n");
    scanf("%s",filename);                       /*输入文件名*/
    if((fp=fopen(filename,"w"))==NULL)          /*判断文件是否打开失败*/
    {
        printf("can not open!\npress any key to continue\n");
        getchar();
        exit(0);
    }
    fprintf(fp,"%c",i);                         /*将 88 以字符的形式写入 fp 所指的磁盘文件中*/
    fclose(fp);
}
```

程序运行界面如图 14.8 所示。

将数字 88 以字符的形式写入磁盘文件中，最终的文件效果如图 14.9 所示。

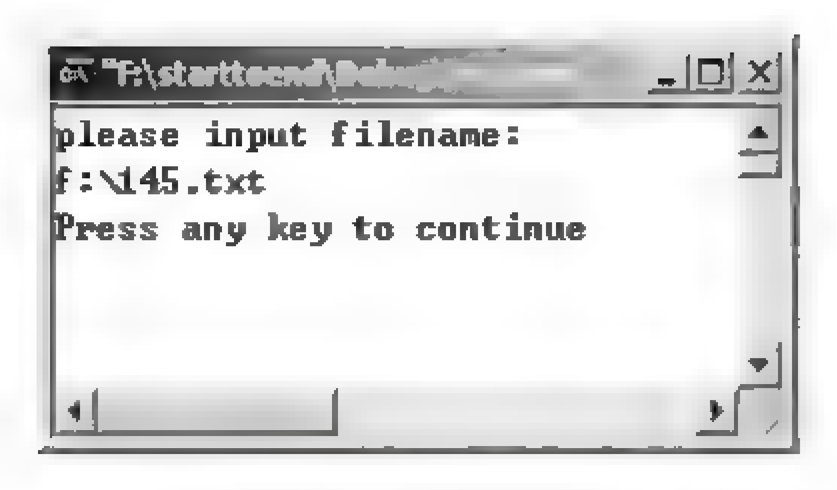


图 14.8　运行界面

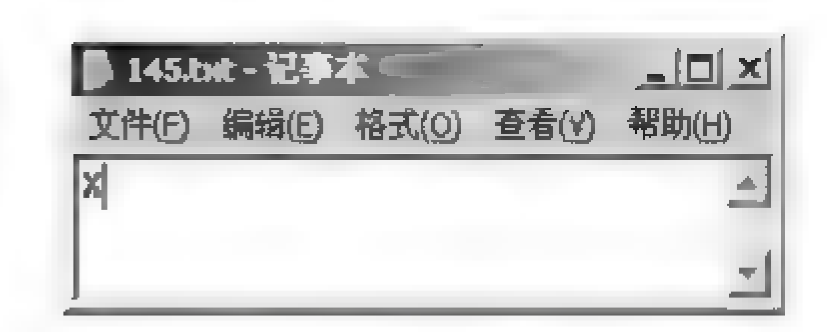


图 14.9　生成的文件效果

14.3.6　fscanf 函数

fscanf 函数的一般形式如下：

```
fscanf(文件类型指针,格式字符串,输入列表);
```

例如：

```
fscanf(fp,"%d",&i);
```

它的作用是读入 fp 所指向的文件中的 i 的值。

【例 14.6】　将文件中的 5 个字符以整数形式输出。（**实例位置：资源包\TM\sl\14\6**）

```
#include<stdio.h>
#include<process.h>
main()
{
    FILE *fp;
    char i,j;
    char filename[30];                          /*定义一个字符型数组*/
    printf("please input filename:\n");
    scanf("%s",filename);                       /*输入文件名*/
    if((fp=fopen(filename,"r"))==NULL)          /*判断文件是否打开失败*/
    {
        printf("can not open!\npress any key to continue\n");
        getchar();
        exit(0);
    }
    for(i=0;i<5;i++)
    {
        fscanf(fp,"%c",&j);
        printf("%d is:%5d\n",i+1,j);
    }
    fclose(fp);
}
```

所读取的磁盘文件中的内容如图 14.10 所示。

程序运行界面如图 14.11 所示。



图 14.10　文件中的内容

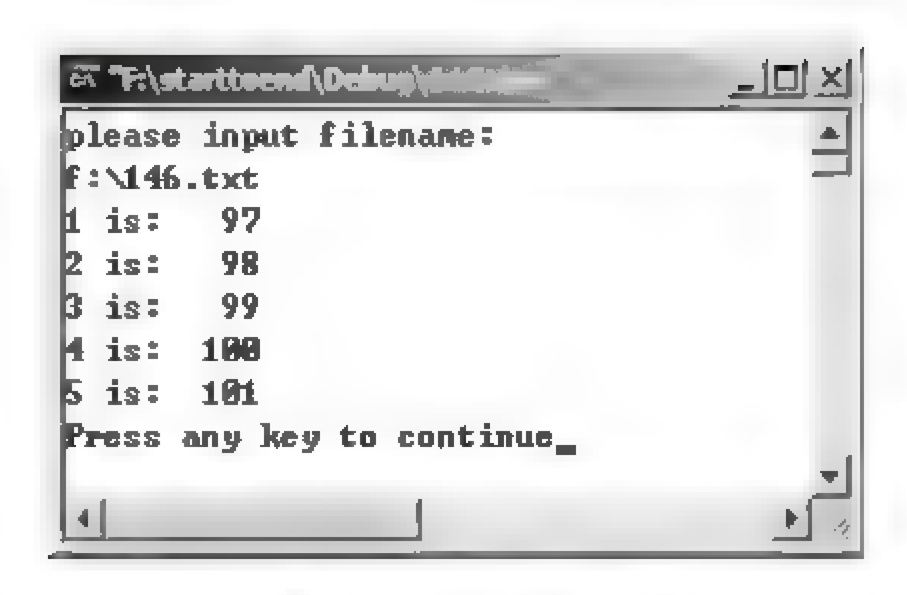


图 14.11　运行界面

14.3.7　fread 和 fwrite 函数

前面介绍的 fputc 和 fgetc 函数，每次只能读写文件中的一个字符，但是在编写程序的过程中往往需要对整块数据进行读写，例如，对一个结构体类型变量值进行读写。下面就介绍实现整块读写功能的 fread 和 fwrite 函数。

fread 函数的一般形式如下：

```
fread(buffer,size,count,fp);
```

该函数的作用是从 fp 所指的文件中读入 count 次，每次读 size 字节，读入的信息存在 buffer 地址中。

fwrite 函数的一般形式如下：

```
fwrite(buffer,size,count,fp);
```

该函数的作用是将 buffer 地址开始的信息输出 count 次，每次写 size 字节到 fp 所指的文件中。

- ☑ buffer：一个指针。对于 fwrite 函数来说，是要输出数据的地址（起始地址）；对 fread 函数来说，是所要读入的数据存放的地址。
- ☑ size：要读写的字节数。
- ☑ count：要读写多少个 size 字节的数据项。
- ☑ fp：文件型指针。

例如：

```
fread(a,2,3,fp);
```

其含义是从 fp 所指的文件中每次读两个字节送入实数组 a 中，连续读 3 次。

```
fwrite(a,2,3,fp);
```

其含义是将 a 数组中的信息每次输出两个字节到 fp 所指向的文件中，连续输出 3 次。

【例 14.7】　编程实现将录入的通讯录信息保存到磁盘文件中，在录入完信息后，将所录入的信息全部显示出来。（**实例位置：资源包\TM\sl\14\7**）

```
#include<stdio.h>
#include<process.h>
struct address_list                                    /*定义结构体，存储学生成绩信息*/
{
```

```
    char name[10];
    char adr[20];
    char tel[15];
} info[100];
void save(char *name, int n)                          /*自定义 save 函数*/
{
    FILE *fp;                                         /*定义一个指向 FILE 类型结构体的指针变量*/
    int i;
    if((fp = fopen(name, "wb")) == NULL)              /*以只写方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(0);
    }
    for(i = 0; i < n; i++)
        if(fwrite(&info[i], sizeof(struct address_list), 1, fp) != 1) /*将一组数据输出到 fp 所指的文件中*/
            printf("file write error\n");             /*如果写入文件不成功，则输出错误*/
    fclose(fp);                                       /*关闭文件*/
}
void show(char *name, int n)                          /*自定义 show 函数*/
{
    int i;
    FILE *fp;                                         /*定义一个指向 FILE 类型结构体的指针变量*/
    if((fp = fopen(name, "rb")) == NULL)              /*以只读方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(0);
    }
    for(i = 0; i < n; i++)
    {
        fread(&info[i], sizeof(struct address_list), 1, fp);   /*从 fp 所指向的文件读入数据存到 score 数组中*/
        printf("%15s%20s%20s\n", info[i].name, info[i].adr,info[i].tel);
    }
    fclose(fp);                                       /*关闭文件*/
}
main()
{
    int i, n;                                         /*变量类型为基本整型*/
    char filename[50];                                /*数组为字符型*/
    printf("how many ?\n");
    scanf("%d", &n);                                  /*输入学生数*/
    printf("please input filename:\n");
    scanf("%s", filename);                            /*输入文件所在路径及名称*/
    printf("please input name,address,telephone:\n");
    for (i = 0; i < n; i++)                           /*输入学生成绩信息*/
    {
        printf("NO%d", i + 1);
        scanf("%s%s%s", info[i].name, info[i].adr, info[i].tel);
        save(filename, n);                            /*调用函数 save*/
    }
```

```
    show(filename, n);                                    /*调用函数 show*/
}
```

程序运行结果如图 14.12 所示。

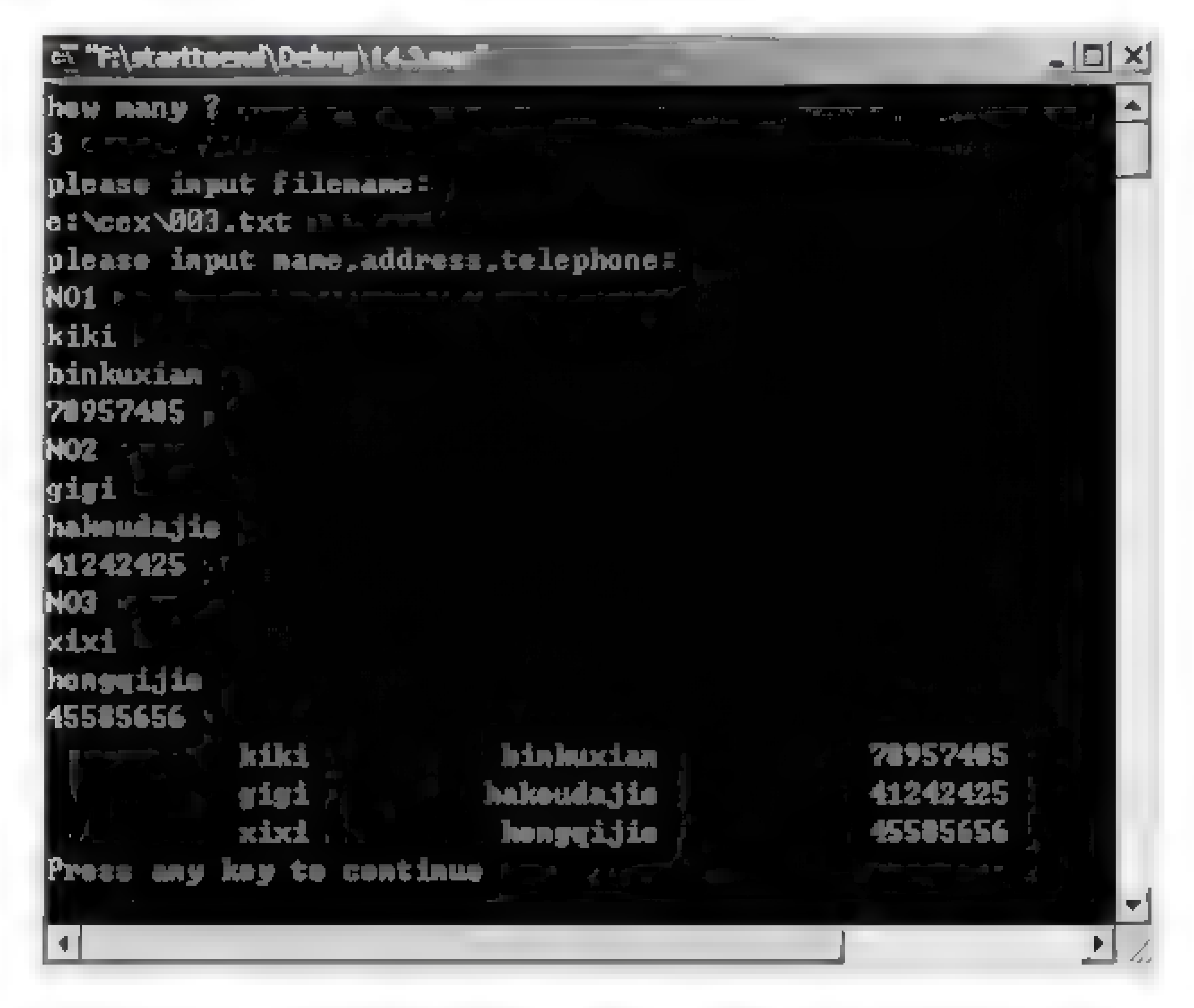


图 14.12　录入并显示信息



14.4　文件的定位

在对文件进行操作时，往往不需要从头开始，一般只需对其中指定的内容进行操作。这时，就需要使用文件定位函数来实现对文件的随机读取。本节将介绍 3 种随机读写函数。

14.4.1　fseek 函数

借助缓冲型 I/O 系统中的 fseek 函数，可以完成随机读写操作。fseek 函数的一般形式如下：

```
fseek(文件类型指针,位移量,起始点);
```

该函数的作用是移动文件内部的位置指针。其中，“文件类型指针”指向被移动的文件；“位移量”表示移动的字节数，要求位移量是 long 型数据，以便在文件长度大于 64KB 时不会出错。当用常量表示位移量时，要求加后缀“L”；“起始点”表示从何处开始计算位移量，规定的起始点有文件首、文件当前位置和文件尾 3 种，其表示方法如表 14.2 所示。

表 14.2　起始点

起 始 点	表 示 符 号	数 字 表 示
文件首	SEEK—SET	0
文件当前位置	SEEK—CUR	1
文件尾	SEEK—END	2

例如：

```
fseek(fp,-20L,1);
```

表示将位置指针从当前位置向后退 20 个字节。



说明

fseek 函数一般用于二进制文件。在文本文件中由于要进行转换，往往计算的位置会出现错误。

文件的随机读写在移动位置指针之后进行，即可用前面介绍的任一种读写函数进行读写。

【例 14.8】 向任意一个二进制文件中写入一个长度大于 6 的字符串，然后从该字符串的第 6 个字符开始，输出余下字符。（**实例位置：资源包\TM\sl\14\8**）

```
#include<stdio.h>
#include<process.h>
main()
{
    FILE *fp;
    char filename[30],str[50];                    /*定义两个字符型数组*/
    printf("please input filename:\n");
    scanf("%s",filename);                         /*输入文件名*/
    if((fp=fopen(filename,"wb"))==NULL)           /*判断文件是否打开失败*/
    {
        printf("can not open!\npress any key to continue\n");
        getchar();
        exit(0);
    }
    printf("please input string:\n");
    getchar();
    gets(str);
    fputs(str,fp);
    fclose(fp);
    if((fp=fopen(filename,"rb"))==NULL)           /*判断文件是否打开失败*/
    {
        printf("can not open!\npress any key to continue\n");
        getchar();
        exit(0);
    }
    fseek(fp,5L,0);
    fgets(str,sizeof(str),fp);
    putchar('\n');
    puts(str);
    fclose(fp);
}
```

程序运行结果如图 14.13 所示。

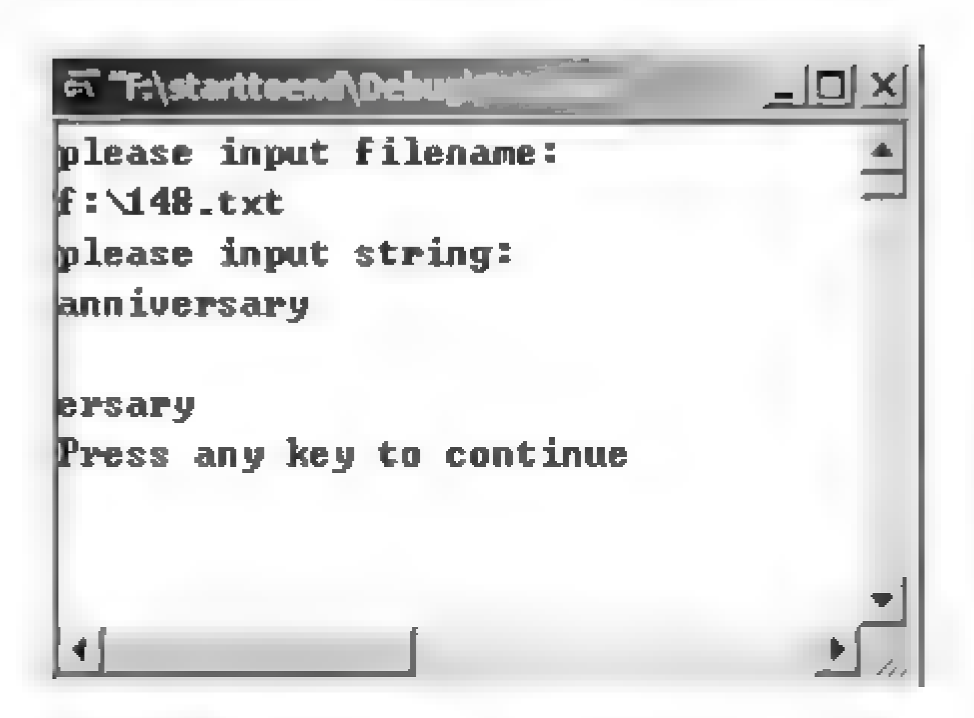


图 14.13　fseek 函数的应用

程序中有这样一句代码：

```
fseek(fp,5L,0);
```

此代码的含义是将文件指针指向距文件首 5 个字节的位置，也就是指向字符串中的第 6 个字符。

14.4.2　rewind 函数

前面讲过了 fseek 函数，这里将要介绍的 rewind 函数也能起到定位文件指针的作用，从而达到随机读写文件的目的。rewind 函数的一般形式如下：

```
int rewind(文件类型指针)
```

该函数的作用是使位置指针重新返回文件的开头，该函数没有返回值。

【例 14.9】　rewind 函数的应用。（**实例位置：资源包\TM\sl\14\9**）

```
#include<stdio.h>
#include<process.h>
main()
{
    FILE *fp;
    char ch,filename[50];
    printf("please input filename:\n");
    scanf("%s",filename);                          /*输入文件名*/
    if((fp=fopen(filename,"r"))==NULL)             /*以只读方式打开该文件*/
    {
        printf("cannot open this file.\n");
        exit(0);
    }
    ch = fgetc(fp);
    while(ch != EOF)
    {
        putchar(ch);                               /*输出字符*/
        ch = fgetc(fp);                            /*获取 fp 指向文件中的字符*/
    }
    rewind(fp);                                    /*指针指向文件开头*/
```

```
        ch = fgetc(fp);
        while(ch != EOF)
        {
            putchar(ch);                                /*输出字符*/
            ch = fgetc(fp);
        }
        fclose(fp);                                     /*关闭文件*/
}
```

程序运行结果如图 14.14 所示。

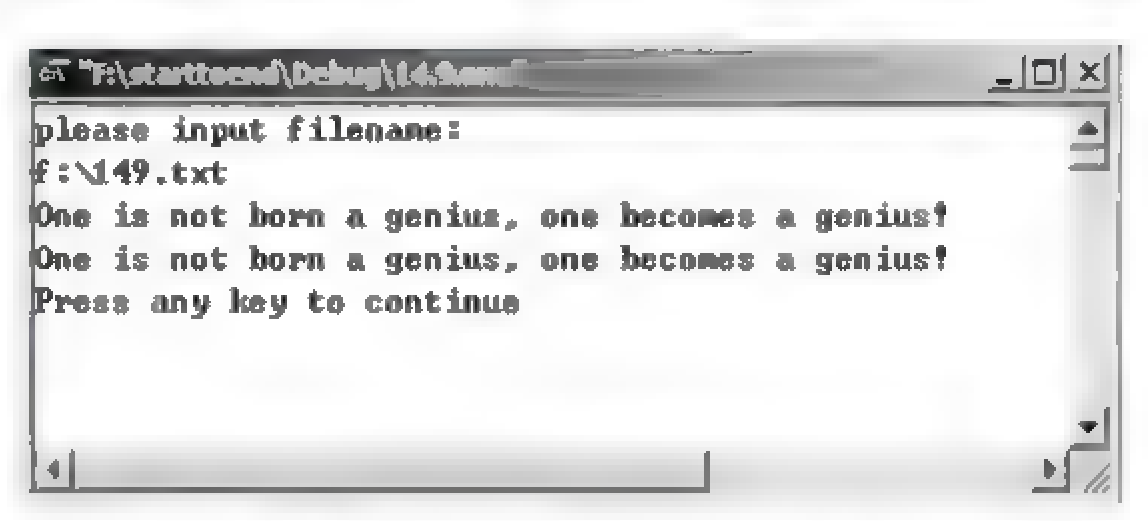


图 14.14　rewind 函数的应用

程序中通过以下 6 行语句，输出第一个“One is not born a genius, one becomes a genius!”。

```
ch = fgetc(fp);
    while(ch != EOF)
    {
        putchar(ch);
        ch = fgetc(fp);
    }
```

在输出第一个“One is not born a genius, one becomes a genius!”后，文件指针已经移动到了该文件的尾部，使用 rewind 函数再次将文件指针移到文件的开始部分，因此当再次使用上面 6 行语句时，就输出了第二个“One is not born a genius, one becomes a genius!”。

14.4.3　ftell 函数

ftell 函数的一般形式如下：

```
long ftell(文件类型指针)
```

该函数的作用是得到流式文件中的当前位置，用相对于文件开头的位移量来表示。当 ftell 函数的返回值为-1L 时，表示出错。

【例 14.10】　求字符串长度。（**实例位置：资源包\TM\sl\14\10**）

```
#include<stdio.h>
#include<process.h>
main()
{
        FILE *fp;
```

```
    int n;
    char ch,filename[50];
    printf("please input filename:\n");
    scanf("%s",filename);                          /*输入文件名*/
    if((fp=fopen(filename,"r"))==NULL)             /*以只读方式打开该文件*/
    {
        printf("cannot open this file.\n");
        exit(0);
    }
    ch = fgetc(fp);
    while(ch != EOF)
    {
        putchar(ch);                               /*输出字符*/
        ch = fgetc(fp);                            /*获取 fp 指向文件中的字符*/
    }
    n=ftell(fp);
    printf("\nthe length of the string is:%d\n",n);
    fclose(fp);                                    /*关闭文件*/
}
```

程序运行结果如图 14.15 所示。

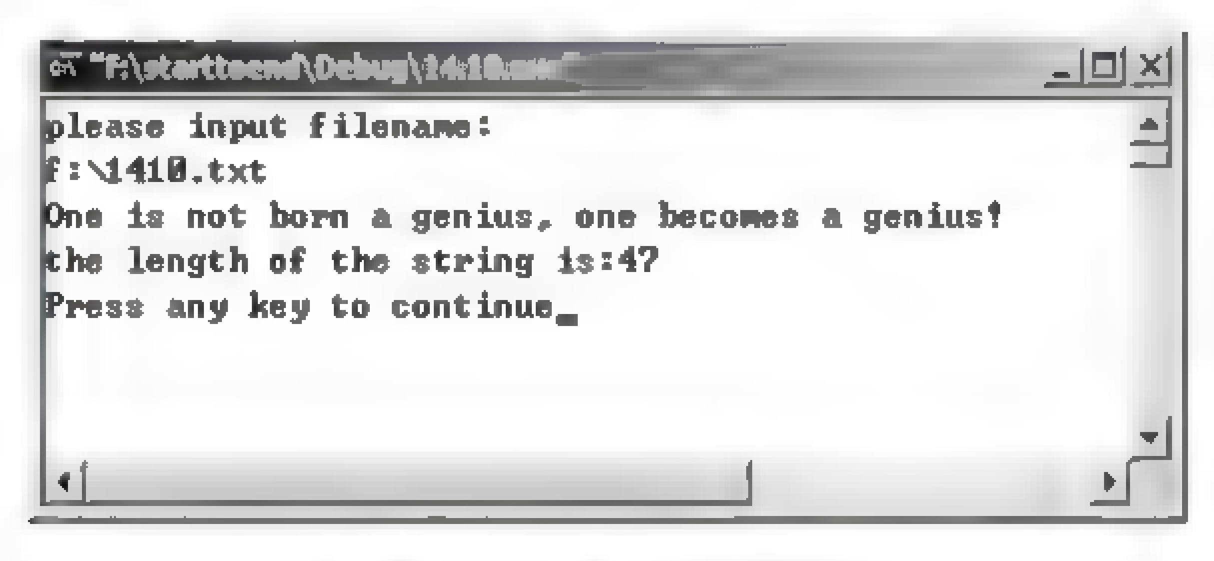


图 14.15　求字符串长度

本节主要讲解了 fseek、rewind 及 ftell 函数，在编写程序的过程中经常会使用到文件定位函数，例如下面将要介绍的例 14.11，要实现将一个文件中的内容复制到另一个文件中时，就可以使用 fseek 函数直接将文件指针指向文件尾，这样就可以将另一个文件中的内容逐个写到该文件中所有内容的后面，从而实现复制操作。当然，文件的复制操作还有很多其他方法可以实现，但使用 fseek 函数可使代码更简洁。

【例 14.11】　编程实现将文件 2 中的内容复制到文件 1 中。（**实例位置：资源包\TM\sl\14\11**）

```
#include<stdio.h>
#include<process.h>
main()
{
    FILE *fp1,*fp2;
    char ch,filename1[30],filename2[30];
    printf("请输入文件 1 的名字：\n");
    scanf("%s",filename1);
    printf("请输入文件 2 的名字：\n");
```

```
    scanf("%s",filename2);
    if((fp1=fopen(filename1,"ab+"))==NULL)
    {
        printf("can not open,press any key to continue\n");
        getchar();
        exit(0);
    }
    if((fp2=fopen(filename2,"rb"))==NULL)
    {
        printf("can not open,press any key to continue\n");
        getchar();
        exit(0);
    }
    fseek(fp1,0L,2);
    while((ch=fgetc(fp2))!=EOF)
    {
        fputc(ch,fp1);
    }
    fclose(fp1);
    fclose(fp2);
}
```

程序运行结果如图 14.16 所示。

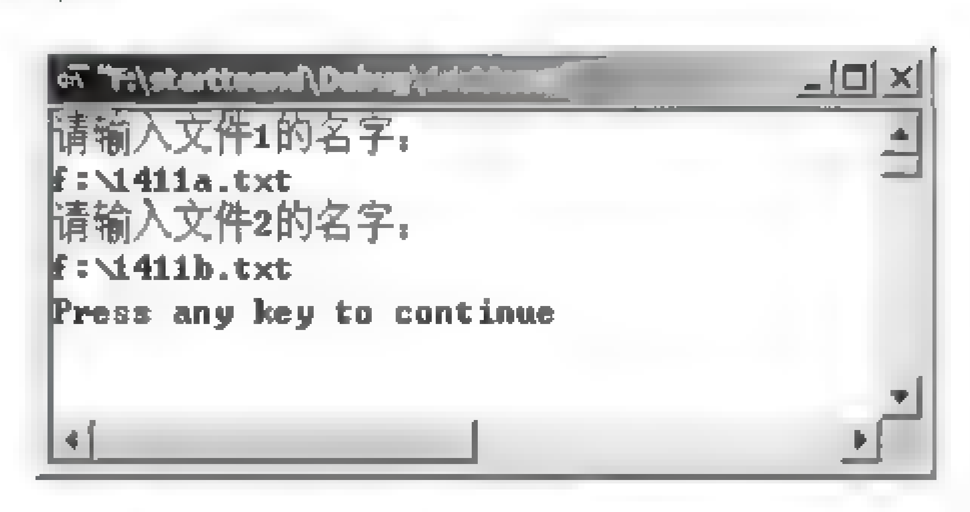


图 14.16　输入要进行复制操作的文件

未进行复制前，两文件中的内容分别如图 14.17 和图 14.18 所示。

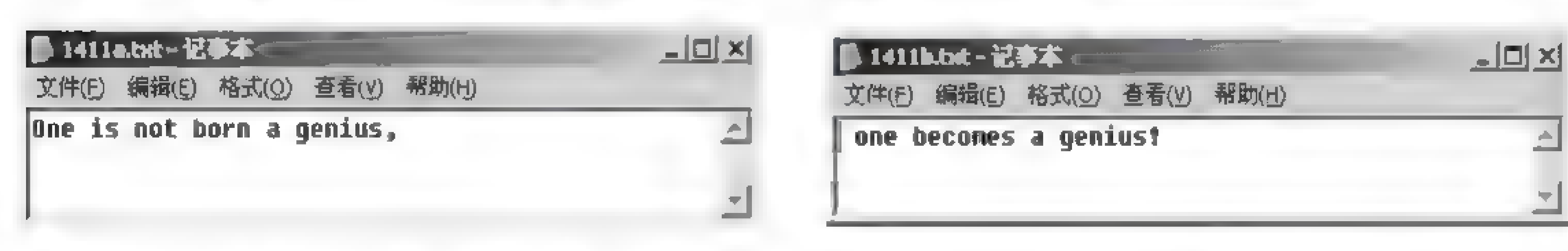


图 14.17　文件 1 中的内容　　　　图 14.18　文件 2 中的内容

复制操作完成后，文件 1 中的内容如图 14.19 所示。

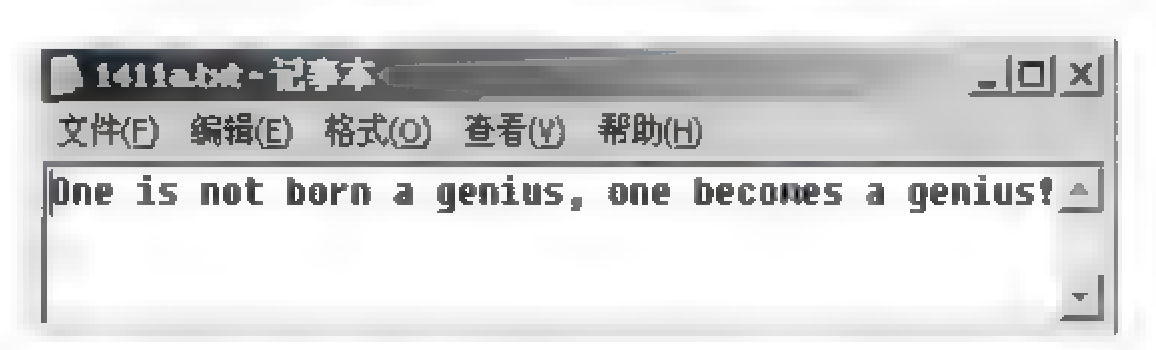


图 14.19　执行复制操作后文件 1 中的内容

14.5　小　　结

本章主要介绍了对文件的一些基本操作，包括文件的打开、关闭、文件的读写及定位等。C 文件按编码方式分为二进制文件和 ASCII 文件。C 语言用文件指针标识文件，文件在读写操作之前必须打开，读写结束必须关闭。文件可以采用不同方式打开，同时必须指定文件的类型。文件的读写也分为多种方式，本章提到了单个字符的读写、字符串的读写、成块读写，以及按指定的格式读写。文件内部的位置指针可指示当前的读写位置，同时也可以移动该指针，从而实现对文件的随机读写。

14.6　实践与练习

1．将一个已存在的文本文档的内容复制到新建的文本文档中。**（答案位置：资源包\TM\sl\14\12）**

2．输入学生人数以及每个学生的数学、语文、英语成绩，并将输入的内容保存到磁盘文件中。**（答案位置：资源包\TM\sl\14\13）**

第15章

存储管理

（视频讲解：22分钟）

程序在运行时，将需要的数据都组织存放在内存空间中，以备程序使用。在软件开发过程中，常常需要动态地分配和撤销内存空间。例如，对动态链表中的节点进行插入和删除，就要对内存进行管理。

本章致力于使读者了解内存的组织结构，了解堆和栈的区别，掌握使用动态管理内存的函数，了解内存在什么情况下会丢失。

通过阅读本章，您可以：

- 了解内存组织方式
- 区分堆与栈的不同
- 能够动态管理所用函数
- 了解内存丢失情况

15.1 内存组织方式



程序存储的概念是当代所有数字计算机的基础，程序的机器语言指令和数据都存储在同一个逻辑内存空间里。

在讲述有关链表的内容时，曾提及动态分配内存的有关函数。那么这些内存是按照怎样的方式组织的呢？下面将会进行具体的介绍。

15.1.1 内存的组织方式

开发人员将程序编写完成之后，程序要先装载到计算机的内核或者半导体内存中，再运行程序。程序被组织成以下 4 个逻辑段：

☑ 可执行代码。

☑ 静态数据。可执行代码和静态数据存储在固定的内存位置。

☑ 动态数据（堆）。程序请求动态分配的内存来自内存池，也就是上面所列举的堆。

☑ 栈。局部数据对象、函数的参数，以及调用函数和被调用函数的联系放在称为栈的内存池中。

以上 4 类根据操作平台和编译器的不同，堆和栈既可以是被所有同时运行的程序共享的操作系统资源，也可以是使用程序独占的局部资源。

15.1.2 堆与栈

通过内存组织方式可以看到，堆用来存放动态分配内存空间，而栈用来存放局部数据对象、函数的参数，以及调用函数和被调用函数的联系，下面对二者进行详细的说明。

1. 堆

在内存的全局存储空间中，用于程序动态分配和释放的内存块称为自由存储空间，通常也称之为堆。在 C 程序中，用 malloc 和 free 函数来从堆中动态地分配和释放内存。

【例 15.1】 在堆中分配内存并释放。（**实例位置：资源包\TM\sl\15\1**）

在本实例中，使用 malloc 函数分配一个整型变量的内存空间，在使用完该空间后，使用 free 函数进行释放。

```
#include<stdio.h>

int main()
{
    int *pInt;                              /*定义整型指针*/
    pInt=(int*)malloc(sizeof(int));         /*分配内存*/

    *pInt=100;                              /*使用分配内存*/
```

```
    printf("the number is:%d\n",*pInt);              /*输出显示数值*/
    free(pInt);                                       /*释放内存*/
    return 0;
}
```

在本程序中，使用 malloc 函数分配一个整型变量的内存空间。

运行程序，显示效果如图 15.1 所示。

2．栈

程序不会像处理堆那样，在栈中显式地分配内存。当程序调用函数和声明局部变量时，系统将自动分配内存。

栈是一个后进先出的压入弹出式的数据结构。在程序运行时，需要每次向栈中压入一个对象，然后栈指针向下移动一个位置。当系统从栈中弹出一个对象时，最晚进栈的对象将被弹出，然后栈指针向上移动一个位置。如果栈指针位于栈顶，则表示栈是空的；如果栈指针指向最下面的数据项的后一个位置，则表示栈为满的。其过程如图 15.2 所示。



图 15.1　在堆中分配内存并释放

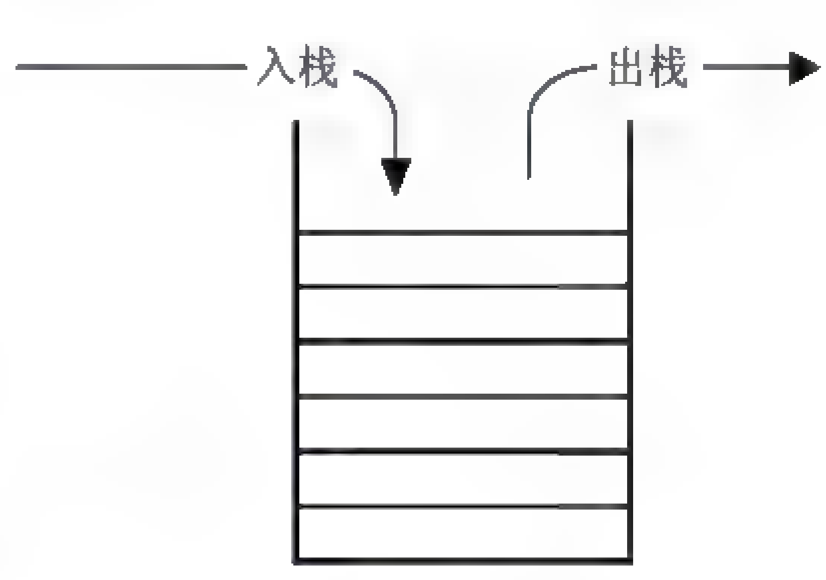


图 15.2　栈操作

程序员经常会利用栈这种数据结构来处理那些最适合用后进先出逻辑来描述的编程问题。这里讨论的栈在程序中都会存在，它不需要程序员编写代码去维护，而是运行时由系统自动处理。所谓的运行时系统维护，实际上就是编译器所产生的程序代码。尽管在源代码中看不到它们，但程序员应该对此有所了解。这个特性和后进先出的特性是栈明显区别于堆的标志。

那么栈是如何工作的呢？例如，当一个函数 A 调用另一个函数 B 时，系统将会把函数 A 的所有实参和返回地址压入栈中，栈指针将移到合适的位置来容纳这些数据。最后进栈的是函数 A 的返回地址。

当函数 B 开始执行后，系统把函数 B 的自变量压入栈中，并把栈指针再向下移，以保证有足够的空间来存储函数 B 声明的所有自变量。

当函数 A 的实参压入栈后，函数 B 就在栈中以自变量的形式建立了形参。函数 B 内部的其他自变量也是存放在栈中的。由于这些进栈操作，栈指针已经移到所有局部变量之下。但是函数 B 记录了刚开始执行时的初始栈指针，以这个指针为参考，用正偏移量或负偏移量来访问栈中的变量。

当函数 B 准备返回时，系统弹出栈中的所有自变量，这时栈指针移到了函数 B 刚开始执行时的位置。接着，函数 B 返回，系统从栈中弹出返回地址，函数 A 就可以继续执行了。

当函数 A 继续执行时，系统还能从栈中弹出调用者的实参，于是栈指针又回到了调用发生前的位置。

【例 15.2】　栈在函数调用时的操作。（**实例位置：资源包\TM\sl\15\2**）

在本实例中，对上面栈的描述操作过程使用实例进行说明。其中函数的名称根据上面描述所确定。

该实例有助于更好地理解栈的操作过程。

```
#include<stdio.h>

void DisplayB(char* string)                    /*函数 B*/
{
    printf("%s\n",string);
}

void DisplayA(char* string)                    /*函数 A*/
{
    char String[20]="LoveWorld!";
    printf("%s\n",string);
    DisplayB(String);                          /*调用函数 B*/
}

int main()
{
    char String[20]="LoveChina!";
    DisplayA(String);                          /*将参数传入函数 A 中*/
    return 0;
}
```

在本程序中，定义函数 A 和 B，其中在函数 A 中再次调用函数 B。根据栈的原理移动栈中的指针，进而存储数据。

运行程序，显示效果如图 15.3 所示。



图 15.3　栈在函数调用时的操作

15.2　动 态 管 理



15.2.1　malloc 函数

malloc 函数的原型如下：

```
void *malloc(unsigned int size);
```

stdlib.h 头文件包含该函数，其作用是在内存中动态地分配一块 size 大小的内存空间。malloc 函数会返回一个指针，该指针指向分配的内存空间，如果出现错误，则返回 NULL。

注意

使用 malloc 函数分配的内存空间位于堆中，而不是在栈中。因此，在使用完这块内存之后，一定要将其释放掉，释放内存空间使用的是 free 函数（下面将会进行介绍）。

例如，使用 malloc 函数分配一个整型内存空间：

```
int *pInt;
pInt=(int*)malloc(sizeof(int));
```

首先定义指针 pInt 用来保存分配内存的地址。在使用 malloc 函数分配内存空间时，需要指定具体的内存空间的大小（size），这时调用 sizeof 函数就可以得到指定类型的大小。malloc 函数成功分配内存空间后会返回一个指针，因为分配的是一个 int 型空间，所以在返回指针时也应该是相对应的 int 型指针，这样就要进行强制类型转换。最后将函数返回的指针赋值给指针 pInt，就可以保存动态分配的整型空间地址了。

【例 15.3】　使用 malloc 函数动态分配空间。（**实例位置：资源包\TM\sl\15\3**）

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
	int* iIntMalloc=(int*)malloc(sizeof(int));		/*分配空间*/
	*iIntMalloc=100;					/*使用该空间保存数据*/
	printf("%d\n",*iIntMalloc);				/*输出数据*/
	return 0;
}
```

在程序中使用 malloc 函数分配了内存空间，通过指向该内存空间的指针，使用该空间保存数据，最后显示该数据，表示保存数据成功。

运行程序，显示效果如图 15.4 所示。



图 15.4　使用 malloc 函数动态分配空间

15.2.2　calloc 函数

calloc 函数的原型如下：

```
void * calloc(unsigned n, unsigned size);
```

使用该函数也要包含头文件 stdlib.h，其功能是在内存中动态分配 n 个长度为 size 的连续内存空间数组。calloc 函数会返回一个指针，该指针指向动态分配的连续内存空间地址。当分配空间错误时，返回 NULL。

例如，使用该函数分配一个整型数组内存空间：

```
int* pArray;                                    /*定义指针*/
pArray=(int*)calloc(3,sizeof(int));             /*分配内存数组*/
```

上面代码中的 pArray 为一个整型指针，使用 calloc 函数分配内存数组，在参数中第一个参数表示分配数组中元素的个数，而第二个参数表示元素的类型。最后将返回的指针赋予 pArray 指针变量，pArray 指向的就是该数组的首地址。

【例 15.4】 使用 calloc 函数分配数组内存。（**实例位置：资源包\TM\sl\15\4**）

在本实例中，动态分配一个数组。使用循环为数组中的每一个元素进行赋值，再将数组中的元素值进行输出，验证分配内存，正确保存数据。

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int* pArray;                                /*定义指针*/
    int i;                                      /*循环控制变量*/
    pArray=(int*)calloc(3,sizeof(int));         /*数组内存*/

    for(i=1;i<4;i++)                            /*使用循环对数组进行赋值*/
    {
        *pArray=10*i;                           /*赋值*/
        printf("NO%d is: %d\n",i,*pArray);      /*显示结果*/
        pArray+=1;                              /*移动指针到数组的下一个元素*/
    }
    return 0;
}
```

在代码中可以看到，使用 calloc 函数分配一个整型数组空间具有 3 个元素，使用 pArray 得到该空间的首地址，因为首地址即为第一个元素的地址，所以通过该指针可以直接输出第一个元素的数据。通过移动指针指向数组中其他的元素，然后将其显示输出。

运行程序，显示效果如图 15.5 所示。

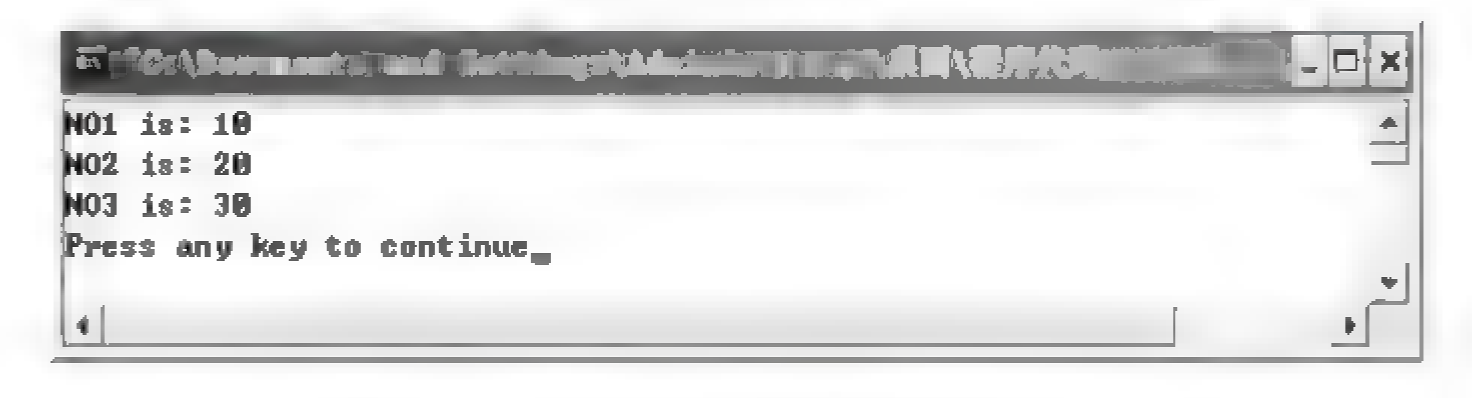


图 15.5 使用 calloc 函数分配数组内存

15.2.3 realloc 函数

realloc 函数的原型如下：

```
void *realloc(void *ptr, size_t size);
```

使用该函数前，要先包含头文件 stdlib.h，其功能是改变 ptr 指针指向的空间大小为 size 大小。设定的 size 大小可以是任意的，也就是说，既可以比原来的数值大，也可以比原来的数值小。返回值是一个指向新地址的指针，如果出现错误，则返回 NULL。

例如，改变一个分配的实型空间大小成为整型大小：

```
fDouble=(double*)malloc(sizeof(double));
iInt=realloc(fDouble,sizeof(int));
```

其中，fDouble 是指向分配的实型空间，之后使用 realloc 函数改变 fDouble 指向的空间的大小，将其大小设置为整型，然后将改变后的内存空间的地址返回赋值给 iInt 整型指针。

【例 15.5】 使用 realloc 函数重新分配内存。（**实例位置：资源包\TM\sl\15\5**）

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    double *fDouble;                              /*定义实型指针*/
    int* iInt;                                    /*定义整型指针*/
    fDouble=(double*)malloc(sizeof(double));      /*使用 malloc 函数分配实型空间*/
    printf("%d\n",sizeof(*fDouble));              /*输出空间的大小*/
    iInt=realloc(fDouble,sizeof(int));            /*使用 realloc 函数改变分配空间的大小*/
    printf("%d\n",sizeof(*iInt));
    return 0;
}
```

在本实例中，首先使用 malloc 函数分配了一个实型大小的内存空间，然后通过 sizeof 函数输出内存空间的大小，最后使用 realloc 函数得到新的内存空间大小，并输出新空间的大小。比较两者的数值，可以看出新空间与原来的空间大小不一样。

运行程序，显示效果如图 15.6 所示。



图 15.6 使用 realloc 函数重新分配内存

15.2.4 free 函数

free 函数的原型如下：

```
void free(void *ptr);
```

free 函数的功能是释放由指针 ptr 指向的内存区，使部分内存区能被其他变量使用。ptr 是最近一次

调用 calloc 或 malloc 函数时返回的值。free 函数无返回值。

例如，释放一个分配整型变量的内存空间：

```
free(pInt);
```

代码中的 pInt 为一个指向一个整型大小的内存空间，使用 free 函数将其进行释放。

【例 15.6】　使用 free 函数释放内存空间。（**实例位置：资源包\TM\sl\15\6**）

在本实例中，将分配的内存进行释放，并且释放前输出一次内存中保存的数据，释放后再利用指针输出一次。观察两次的结果，可以看出，调用 free 函数之后内存被释放了。

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
	int* pInt;                              /*整型指针*/
	pInt=(int*)malloc(sizeof(pInt));        /*分配整型空间*/
	*pInt=100;                              /*赋值*/
	printf("%d\n",*pInt);                   /*将值进行输出*/
	free(pInt);                             /*释放该内存空间*/
	printf("%d\n",*pInt);                   /*将值进行输出*/
	return 0;
}
```

在程序中定义指针 pInt 用来指向动态分配的内存空间，使用新空间保存数据，之后利用指针进行输出。调用 free 函数将其空间释放，当再输出时因为保存数据的空间已经被释放，所以数据肯定就不存在了。

运行程序，显示效果如图 15.7 所示。

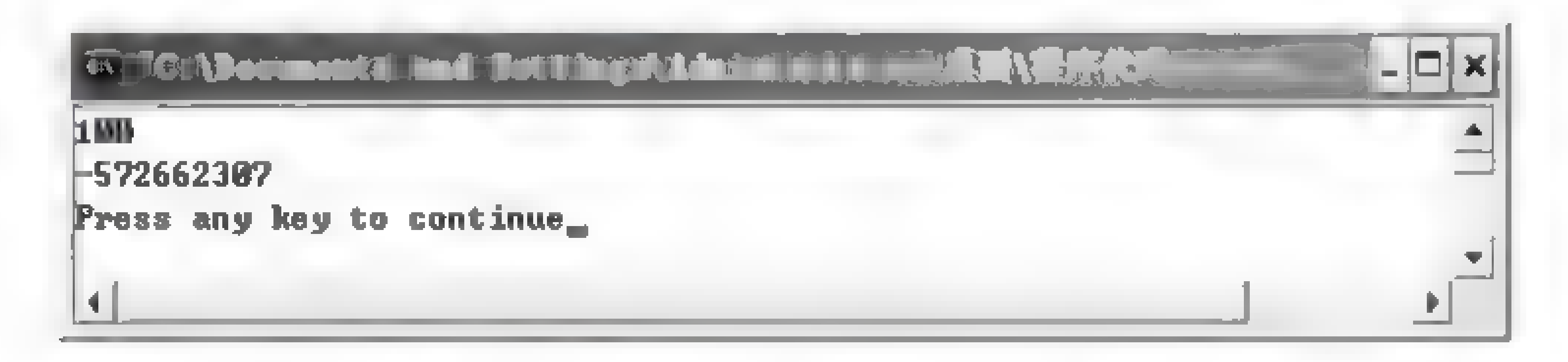


图 15.7　使用 free 函数释放内存空间

15.3　内 存 丢 失



在使用 malloc 等函数分配过内存后，还需要使用 free 函数释放内存。内存不进行释放，会造成内存遗漏，甚至可能会导致系统崩溃。

free 函数的用处在于实时地执行回收内存的操作，如果程序很简单，程序结束之前也不会使用过多的内存，不会降低系统的性能，那么也可以不用写 free 函数去释放内存。程序结束后，操作系统会完成释放的功能。

但是在开发大型程序时如果不写 free 函数去释放内存，后果是很严重的。这是因为很可能在程序

中要重复一万次分配 10MB 的内存，如果每次进行分配内存后都使用 free 函数去释放用完的内存空间，那么这个程序只需要使用 10MB 内存就可以运行。但是如果不使用 free 函数，那么程序就要使用 100GB 的内存！这其中包括绝大部分的虚拟内存，而由于虚拟内存的操作需要读写磁盘，这样会极大地影响到系统的性能，系统因此可能崩溃。

因此，在程序中编写 malloc 函数分配内存后，应对应地写出一个 free 函数释放内存。这是一个良好的编程习惯，不但在处理大型程序时非常有必要，也在一定程度上体现了程序优美的风格和健壮性。

下面来看一个将内存丢失的例子。例如：

```
pOld=(int*)malloc(sizeof(int));
pNew=(int*)malloc(sizeof(int));
```

这两段代码分别表示创建了一块内存，并且将内存的地址传给了指针 pOld 和 pNew，此时指针 pOld 和 pNew 分别指向两块内存。如果进行这样的操作：

```
pOld=pNew;
```

pOld 指针就指向了 pNew 指向的内存地址，这时再进行释放内存操作：

```
free(pOld);
```

此时释放 pOld 所指向的内存空间是原来 pNew 指向的，于是这块空间被释放了。但是 pOld 原来指向的那块内存空间还没有被释放（因为没有指针指向这块内存），所以这块内存就造成了丢失。

15.4 小　　结

本章主要对内存分配问题进行整体的介绍。读者学习了内存的组织方式后，可在编写程序时知道这些空间都是如何进行分配的。

之后讲解有关堆和栈的概念，其中栈式数据结构的主要特性是后进入栈的元素先出，即后进先出。

动态管理包括 malloc、calloc、realloc 和 free 4 个函数，其中 free 函数是用来释放内存空间的。

本章的最后介绍了有关内存丢失的问题，其中要求在编写程序时使用 malloc 函数分配内存的同时要对应写出一个 free 函数来。

15.5 实践与练习

1．要求设计一个程序，为一个具有 3 个元素的数组动态分配内存，为元素赋值并将其输出。程序结束之前将内存空间释放。（**答案位置：资源包\TM\sl\15\7**）

2．要求设计一个程序，为二维数组动态分配内存并且释放内存空间。（**答案位置：资源包\TM\sl\15\8**）

第16章

网络套接字编程

（视频讲解：39分钟）

网络已经遍及生活的每一个角落，存在于人们生活的每一天，这说明网络越来越重要，而学习编写网络应用程序也是学习编程的一部分。网络程序的实现可以有多种方式，Windows Socket就是其中一种比较简单的实现方法。

本章致力于使读者了解有关计算机网络的基础知识，其中包括IP地址、OSI七层参考模型、地址解析、域名系统、TCP/IP协议和端口。详细介绍套接字的有关内容，使读者了解使用套接字编写程序的过程，并且通过实践加深对套接字编写网络应用程序的印象。

通过阅读本章，您可以：

- 了解计算机网络的基本知识
- 了解套接字的概述
- 掌握套接字（socket）编程
- 掌握套接字函数的使用方法
- 使用套接字编写网络应用程序



16.1 计算机网络基础

计算机网络是计算机和通信技术相结合的产物，它代表了计算机发展的重要方向。了解计算机的网络结构有助于用户开发网络应用程序。本节将介绍有关计算机网络的基础知识和基本概念。

16.1.1 IP 地址

为了使网络上的计算机能够彼此识别对方，每台计算机都需要一个 IP 地址以标识自己。IP 地址由 IP 协议规定的 32 位的二进制数表示，最新的 IPv6 协议将 IP 地址升为 128 位，这使得 IP 地址更加广泛，能够很好地解决目前 IP 地址紧缺的情况，但是 IPv6 协议距离实际应用还有一段距离。目前，多数操作系统和应用软件都是以 32 位的 IP 地址为基准。

32 位的 IP 地址主要分为前缀和后缀两部分。前缀表示计算机所属的物理网络，后缀确定该网络上的唯一一台计算机。在互联网上，每一个物理网络都有唯一的网络号，根据网络号的不同，可以将 IP 地址分为 5 类，即 A 类、B 类、C 类、D 类和 E 类。其中，A 类、B 类和 C 类属于基本类，D 类用于多播发送，E 类属于保留。表 16.1 描述了各类 IP 地址的范围。

表 16.1　各类 IP 地址的范围

类　型	范　围
A 类	0.0.0.0～127.255.255.255
B 类	128.0.0.0～191.255.255.255
C 类	192.0.0.0～223.255.255.255
D 类	224.0.0.0～239.255.255.255
E 类	240.0.0.0～247.255.255.255

在上述 IP 地址中，有几个 IP 地址是特殊的，有其单独的用途。

☑ 网络地址：在 IP 地址中主机地址为 0 的表示网络地址，如 128.111.0.0。

☑ 广播地址：在网络号后跟所有位全是 1 的 IP 地址，表示广播地址。

☑ 回送地址：127.0.0.1 表示回送地址，用于测试。

16.1.2 OSI 七层参考模型

开放系统互联（Open System Interconnection，OSI）是国际标准化组织（ISO）为了实现计算机网络的标准化而颁布的参考模型。OSI 参考模型采用分层的划分原则，将网络中的数据传输划分为 7 层，每一层使用下层的服务，并向上层提供服务。表 16.2 描述了 OSI 参考模型的结构。

表 16.2　OSI 参考模型

层　次	名　称	功能描述
第 7 层	应用层（Application）	应用层负责网络中应用程序与网络操作系统之间的联系。例如，建立和结束使用者之间的连接，管理建立相互连接使用的应用资源
第 6 层	表示层（Presentation）	表示层用于确定数据交换的格式，它能够解决应用程序之间在数据格式上的差异，并负责设备之间所需要的字符集和数据的转换
第 5 层	会话层（Session）	会话层是用户应用程序与网络层的接口，它能够建立与其他设备的连接，即会话，并且它能够对会话进行有效的管理
第 4 层	传输层（Transport）	传输层提供会话层和网络层之间的传输服务，该服务从会话层获得数据，必要时对数据进行分割，然后传输层将数据传递到网络层，并确保数据能正确无误地传送到网络层
第 3 层	网络层（Network）	网络层能够将传输的数据封包，然后通过路由选择、分段组合等控制，将信息从源设备传送到目标设备
第 2 层	数据链路层（Data Link）	数据链路层主要是修正传输过程中的错误信号，它能够提供可靠的通过物理介质传输数据的方法
第 1 层	物理层（Physical）	利用传输介质为数据链路层提供物理连接，它规范了网络硬件的特性、规格和传输速度

OSI 参考模型的建立，不仅创建了通信设备之间的物理通道，还规划了各层之间的功能，为标准化组合和生产厂家制定协议提供了基本原则，这有助于用户了解复杂的协议，如 TCP/IP、X.25 协议等。用户可以将这些协议与 OSI 参考模型对比，从而了解这些协议的工作原理。

16.1.3　地址解析

所谓地址解析，是指将计算机的协议地址解析为物理地址，即 MAC（Medium Access Control）地址，又称为媒体访问控制地址。通常，在网络上由地址解析协议（ARP）来实现地址解析。下面以本地网络上的两台计算机通信为例介绍 ARP 协议解析地址的过程。

假设主机 A 和主机 B 处于同一个物理网络上，主机 A 的 IP 为 192.168.1.21，主机 B 的 IP 为 192.168.1.23，当主机 A 与主机 B 进行通信时，主机 B 的 IP 地址 192.168.1.23 将按如下步骤解析为物理地址。

（1）主机 A 从本地 ARP 缓存中查找 IP 为 192.168.1.23 对应的物理地址。用户可以在命令行窗口中输入“arp -a”命令查看本地 ARP 缓存，如图 16.1 所示。



图 16.1　本地 ARP 缓存

（2）如果主机 A 在 ARP 缓存中没有发现 192.168.1.23 映射的物理地址，将发送 ARP 请求帧到本地网络上的所有主机，在 ARP 请求帧中包含了主机 A 的物理地址和 IP 地址。

（3）本地网络上的其他主机接收到 ARP 请求帧后，检查是否与自己的 IP 地址匹配，如果不匹配，则丢弃 ARP 请求帧。如果主机 B 发现与自己的 IP 地址匹配，则将主机 A 的物理地址和 IP 地址添加到自己的 ARP 缓存中，然后主机 B 将自己的物理地址和 IP 地址发送到主机 A，当主机 A 接收到主机 B 发来的信息，将以这些信息更新 ARP 缓存。

（4）当主机 B 的物理地址确定后，主机 A 就可以与主机 B 进行通信了。

16.1.4 域名系统

虽然使用 IP 地址可以标识网络中的计算机，但是 IP 地址容易混淆，并且不容易记忆，人们更倾向于使用主机名来标识 IP 地址。由于在 Internet 上存在许多计算机，为了防止主机名相同，Internet 管理机构采取了在主机名后加上后缀名的方法标识一台主机，其后缀名被称为域名。例如，www.mingrisoft.com，主机名为 www，域名为 mingrisoft.com。这里的域名为二级域名，其中 com 为一级域名，表示商业组织，mingrisoft 为本地域名。为了能够利用域名进行不同主机间的通信，需要将域名解析为 IP 地址，称之为域名解析。域名解析是通过域名服务器来完成的。

假如主机 A 的本地域名服务器是 dns.local.com，根域名服务器是 dns.mr.com；所要访问的主机 B 的域名为 www.mingribook.com，域名服务器为 dns.mrbook.com。当主机 A 通过域名 www.mingribook.com 访问主机 B 时，将发送解析域名 www.mingribook.com 的报文，本地的域名服务器收到请求后，查询本地缓存，假设没有该记录，则本地域名服务器 dns.local.com 向根域名服务器 dns.mr.com 发出请求解析域名 www.mingribook.com。根域名服务器 dns.mr.com 收到请求后查询本地记录，如果发现 mingribook.com NS dns.mrbook.com 信息，将给出 dns.mrbook.com 的 IP 地址，并将结果返回给主机 A 的本地域名服务器 dns.local.com，当本地域名服务器 dns.local.com 收到信息后，会向主机 B 的域名服务器 dns.mrbook.com 发送解析域名 www.mingribook.com 的报文。当域名服务器 dns.mrbook.com 收到请求后，开始查询本地的记录，发现 www.mingribook.com A 211.120.X.X 类似的信息，将结果返回给主机 A 的本地域名服务器 dns.local.com，其中 211.120.X.X 表示域名 www.mingribook.com 的 IP 地址。

16.1.5 TCP/IP 协议

TCP/IP（Transmission Control Protocal/Internet Protocal，传输控制协议/网际协议）是互联网上最流行的协议，它能够实现互联网上不同类型操作系统的计算机相互通信。对于网络开发人员，必须了解 TCP/IP 协议的结构。TCP/IP 协议将网络分为 4 层，分别对应于 OSI 参考模型的 7 层结构。表 16.3 列出了 TCP/IP 协议与 OSI 参考模型的对应关系。

表 16.3　TCP/IP 协议结构层次

TCP/IP 协议	OSI 参考模型
应用层（包括 Telnet、FTP、SNTP 协议）	会话层、表示层和应用层
传输层（包括 TCP、UDP 协议）	传输层
网络层（包括 ICMP、IP、ARP 等协议）	网络层
数据链路层	物理层和数据链路层

由表 16.3 可以发现，TCP/IP 协议不是单个协议，而是一个协议簇，它包含多种协议，其中主要的协议有网际协议（IP）和传输控制协议（TCP）等。下面给出 TCP/IP 主要协议的结构。

1. TCP 协议

传输控制协议（TCP）是一种提供可靠数据传输的通用协议，它是 TCP/IP 体系结构中传输层上的协议。在发送数据时，应用层的数据传输到传输层，加上 TCP 的首部，数据就构成了报文。报文是网际层 IP 的数据，如果再加上 IP 首部，就构成了 IP 数据报。TCP 协议的 C 语言数据描述如下：

```
typedef struct HeadTCP
{
    WORD    SourcePort;         /*16 位源端口号*/
    WORD    DePort;             /*16 位目的端口*/
    DWORD   SequenceNo;         /*32 位序号*/
    DWORD   ConfirmNo;          /*32 位确认序号*/
    BYTE    HeadLen;        /*与 Flag 为一个组成部分，首部长度，占 4 位，保留 6 位，6 位标识，共 16 位*/
    BYTE    Flag;
    WORD    WndSize;            /*16 位窗口大小*/
    WORD    CheckSum;           /*16 位校验和*/
    WORD    UrgPtr;             /*16 位紧急指针*/
} HEADTCP;
```

2. IP 协议

IP 协议又称为网际协议，它工作在网络层，主要提供无链接数据报传输。IP 协议不保证数据报的发送，但可以最大限度地发送数据。IP 协议的 C 语言数据描述如下：

```
typedef struct HeadIP
{
    unsigned char   headerlen:4;    /*首部长度，占 4 位*/
    unsigned char   version:4;      /*版本，占 4 位 */
    unsigned char   servertype;     /*服务类型，占 8 位，即 1 个字节*/
    unsigned short totallen;        /*总长度，占 16 位*/
    unsigned short id;              /*与 idoff 构成标识，共占 16 位，前 3 位是标识，后 13 位是片偏移*/
    unsigned short idoff;
    unsigned char   ttl;            /*生存时间，占 8 位*/
    unsigned char   proto;          /*协议，占 8 位*/
    unsigned short checksum;        /*首部检验和，占 16 位*/
    unsigned int    sourceIP;       /*源 IP 地址，占 32 位*/
    unsigned int    destIP;         /*目的 IP 地址，占 32 位*/
}HEADIP;
```

3. ICMP 协议

ICMP 协议又称为网际控制报文协议。它负责网络上设备状态的发送和报文检查，可以将某个设备的故障信息发送到其他设备上。ICMP 协议的 C 语言数据描述如下：

```
typedef struct HeadICMP
{
    BYTE Type;                      /*8 位类型*/
```

```
    BYTE Code;                    /*8 位代码*/
    WORD ChkSum;                  /*16 位校验和*/
} HEADICMP;
```

4. UDP 协议

用户数据报协议（UDP）是一个面向无连接的协议，采用该协议，两个应用程序不需要先建立连接。它为应用程序提供一次性的数据传输服务。UDP 协议不提供差错恢复，不能提供数据重传，因此该协议传输数据安全性略差。UDP 协议的 C 语言数据描述如下：

```
typedef struct HeadUDP
{
    WORD SourcePort;              /*16 位源端口号*/
    WORD DePort;                  /*16 位目的端口*/
    WORD Len;                     /*16 为 UDP 长度*/
    WORD ChkSum;                  /*16 位 UDP 校验和*/
} HEADUDP;
```

16.1.6 端口

在网络上，计算机是通过 IP 地址来标识自己的，但是当涉及两台计算机具体通信时，还会出现一个问题。假设主机 A 中的应用程序 A1 想与主机 B 中的应用程序 B1 通信，如果知道主机 A 中的是 A1 应用程序与主机 B 中的应用程序通信，而不是主机 A 中的其他应用程序与主机 B 中的应用程序通信，则当主机 B 接收到数据时，它如何知道数据是发往应用程序 B1 的呢？这是因为在主机 B 中可以同时运行多个应用程序。

为了解决上述问题，TCP/IP 协议提出了端口的概念，用于标识通信的应用程序。当应用程序（严格说应该是进程）与某个端口绑定后，系统会将收到的给该端口的数据送往该应用程序。端口是用一个 16 位的无符号整数值来表示的，范围为 0～65535，低于 256 的端口被作为系统的保留端口，用于系统进程的通信，不在这一范围的端口号被称为自由端口，可以由进程自由使用。

16.1.7 套接字的引入

为了更方便地开发网络应用程序，美国的伯克利大学在 UNIX 上推出了一种应用程序访问通信协议的操作系统调用套接字（socket）。socket 的出现，使得程序员可以很方便地访问 TCP/IP，从而开发各种网络应用的程序。后来，套接字被引进到 Windows 等操作系统，成为开发网络应用程序的有效工具。

套接字存在于通信区域中，通信区域也称为地址族，主要用于将通过套接字通信的进程的公有特性综合在一起。套接字通常只与同一区域的套接字交换数据。Windows Sockets 只支持一个通信区域——AF_INET 网际域，使用网际协议族通信的进程使用该域。

16.1.8 网络字节顺序

不同的计算机存放多字节值的顺序不同，有的机器在起始地址存放低位字节，有的机器在起始地

址存放高位字节。基于 Intel CPU 的 PC 机采用低位先存的方式。为了保证数据的正确性，在网络协议中需要指定网络字节顺序，TCP/IP 协议使用 16 位整数和 32 位整数的高位先存格式。由于不同的计算机存放数据字节的顺序不同，这样发送数据后当接收该数据时，也有可能无法查看所接收的数据。因此，在网络中不同主机间进行通信时，要统一采用网络字节顺序。

16.2　套接字基础


视频讲解

套接字是网络通信的基石，是网络通信的基本构件，最初由加利福尼亚大学 Berkeley 分校为 UNIX 开发的网络通信编程接口。为了在 Windows 操作系统上使用套接字，20 世纪 90 年代初，微软和第三方厂商共同制定了一套标准，即 Windows Socket 规范，简称 WinSock。本节将介绍有关 Windows 套接字的相关知识。

16.2.1　套接字概述

所谓套接字，实际上是一个指向传输提供者的句柄。在 WinSock 中，就是通过操作该句柄来实现网络通信和管理的。根据性质和作用的不同，套接字可以分为原始套接字、流式套接字和数据包套接字 3 种。

☑ 原始套接字：是在 WinSock2 规范中提出的，它能够使程序开发人员对底层的网络传输机制进行控制，在原始套接字下接收的数据中包含 IP 头。

☑ 流式套接字：提供双向、有序、可靠的数据传输服务。该类型套接字在通信前需要双方建立连接，大家熟悉的 TCP 协议采用的就是流式套接字。

☑ 数据包套接字：提供双向的数据流，但不能保证数据传输的可靠性、有序性和无重复性。UDP 协议采用的就是数据包套接字。

16.2.2　TCP 的套接字的 socket 编程

TCP 是面向连接的可靠的传输协议。利用 TCP 协议进行通信时，首先要建立通信双方的连接。一旦连接建立完成，就可以进行通信。TCP 提供了数据确认和数据重传的机制，保证了发送的数据一定能到达通信的对方。

基于 TCP 面向连接的 socket 编程的服务器端程序流程如下：

（1）创建套接字 socket。

（2）将创建的套接字绑定（bind）到本地的地址和端口上。

（3）设置套接字的状态为监听状态（listen），准备接受客户端的连接请求。

（4）接受请求（accept），同时返回得到一个用于连接的新套接字。

（5）使用这个新套接字进行通信（通信函数使用 send/recv）。

（6）通信完毕，释放套接字资源（closesocket）。

基于 TCP 面向连接的 socket 编程的客户端程序流程如下：

（1）创建套接字 socket。

（2）向服务器发出连接请求（connect）。

（3）请求连接后与服务器进行通信操作（send/recv）。

（4）释放套接字资源（closesocket）。

在服务器的一端，当调用 accept 函数时（关于套接字函数后文将进行介绍），程序就会进行等待，直到有客户端调用 connect 函数发送连接请求，然后服务器接受该请求，这样服务器与客户端就建立了连接。当两者建立连接后就可以进行通信了。

注意

在服务器端要建立套接字绑定到指定的主机 IP 和端口上等待客户的请求，但是对于客户端来说，当发起连接请求并被接受后，在服务器端就保存了该客户端的 IP 地址和端口号的信息。对于服务器端来说，一旦建立连接，实际上它已经保存了客户端的 IP 地址和端口号的信息了，因此可以利用返回的套接字进行与客户端的通信。

16.2.3　UDP 的套接字的 socket 编程

UDP 是无连接的不可靠的传输协议。采用 UDP 进行通信时，不需要建立连接，可以直接向一个 IP 地址发送数据，但是不能保证对方能收到。

对于基于 UDP 面向无连接的套接字编程来说，服务器端和客户端这种概念不是特别的严格。可以把服务器称为接收端，客户端就是发送数据的发送端。

基于 UDP 面向无连接的 socket 编程的发送端程序流程如下：

（1）创建套接字 socket。

（2）将套接字绑定（bind）到一个本地地址和端口上。

（3）等待接收数据（recvfrom）。

（4）释放套接字资源（closesocket）。

基于 UDP 面向无连接的 socket 编程的接收端程序流程如下：

（1）创建套接字 socket。

（2）向服务器发送数据（sendto）。

（3）释放套接字资源（closesocket）。

注意

在基于 UDP 的套接字编程中，还是需要使用 bind 进行绑定。因为虽然面向无连接的 socket 编程无须建立连接，但是为了完成通信，首先应该启动接收端来接收发送端发送的数据，这样接收端就必须告诉它的地址和端口，才能接收信息。因此，必须调用 bind 函数将套接字绑定到一个本地地址和端口上。

基于 UDP 的套接字编程时，利用的是 sendto 和 recvfrom 两个函数实现数据的发送和接收；而基于 TCP 的套接字编程时，发送数据是调用 send 函数，接收数据使用的是 recv 函数。



16.3　套接字函数

前面介绍了使用套接字编写程序的过程，本节介绍在利用套接字编程时所需要使用的函数。

16.3.1　套接字函数介绍

1. WSAStartup 函数

该函数的功能是初始化套接字库。其原型如下：

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

注意

WSAStartup 函数用于初始化 Ws2_32.dll 动态链接库。在使用套接字函数之前，一定要初始化 Ws2_32.dll 动态链接库。

☑ wVersionRequested：表示调用者使用的 Windows Socket 的版本，高字节记录修订版本，低字节记录主版本。例如，如果 Windows Socket 的版本为 2.1，则高字节记录 1，低字节记录 2。

☑ lpWSAData：是一个 WSADATA 结构指针，该结构详细记录了 Windows 套接字的相关信息。其定义如下：

```
typedef struct WSAData {
    WORD                wVersion;
    WORD                wHighVersion;
    char            szDescription[WSADESCRIPTION_LEN+1];
    char            szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short  iMaxSockets;
    unsigned short  iMaxUdpDg;
    char FAR *      lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

- wVersion：表示调用者使用的 WS2_32.DLL 动态库的版本号。
- wHighVersion：表示 WS2_32.DLL 支持的最高版本，通常与 wVersion 相同。
- szDescription：表示套接字的描述信息，通常没有实际意义。
- szSystemStatus：表示系统的配置或状态信息，通常没有实际意义。
- iMaxSockets：表示最多可以打开多少个套接字。在套接字版本 2 或以后的版本中，该成员将被忽略。
- iMaxUdpDg：表示数据报的最大长度。在套接字版本 2 或以后的版本中，该成员将被忽略。
- lpVendorInfo：表示套接字的厂商信息。在套接字版本 2 或以后的版本中，该成员将被忽略。

例如，使用 WSAStartup 初始化套接字，版本号为 2.2：

```
WORD wVersionRequested;                    /*WORD（字），类型为 unsigned short*/
WSADATA wsaData;                           /*库版本信息结构*/
/*定义版本类型。将两个字节组合成一个字，前面是低字节，后面是高字节*/
wVersionRequested = MAKEWORD(2, 2);        /*表示版本号*/
/*加载套接字库，初始化 Ws2_32.dll 动态链接库*/
WSAStartup(wVersionRequested, &wsaData);
```

从上面的代码中可以看出，MAKEWORD 宏的作用是：根据给定的两个无符号字节，创建一个16 位的无符号整型，将创建的值赋予 wVersionRequested 变量，表示套接字的版本号。

2．socket 函数

该函数的功能是创建一个套接字。其原型如下：

```
SOCKET socket(int af,int type, int protocol);
```

- ☑ af：表示一个地址家族，通常为 AF_INET。
- ☑ type：表示套接字类型。如果为 SOCK_STREAM，表示创建面向连接的流式套接字；如果为 SOCK_DGRAM，表示创建面向无连接的数据报套接字；如果为 SOCK_RAW，表示创建原始套接字。对于这些值，用户可以在 Winsock2.h 头文件中找到。
- ☑ potocol：表示套接字所用的协议，如果用户不指定，可以设置为 0。
- ☑ 返回值：创建的套接字句柄。

例如，使用 socket 函数创建一个套接字 socket_server：

```
/*创建套接字*/
/*AF_INET 表示指定地址族，SOCK_STREAM 表示流式套接字 TCP，特定的地址家族相关的协议*/
socket_server=socket(AF_INET,SOCK_STREAM,0);
```

在代码中，如果 socket 函数调用成功，它就会返回一个新的 SOCKET 数据类型的套接字描述符。使用定义好的套接字 socket_server 进行保存。

3．bind 函数

该函数的功能是将套接字绑定到指定的端口和地址上。其原型如下：

```
int bind(SOCKET s,const struct sockaddr FAR*    name,int namelen);
```

- ☑ s：表示套接字标识。
- ☑ name：是一个 sockaddr 结构指针，该结构中包含了要结合的地址和端口号。
- ☑ namelen：确定 name 缓冲区的长度。
- ☑ 返回值：如果函数执行成功，则返回值为 0，否则为 SOCKET_ERROR。

在创建了套接字之后，应该将该套接字绑定到本地的某个地址和端口上，这时就需要该函数了。例如，使用 bind 函数绑定一个套接字：

```
SOCKADDR_IN Server_add;                    /*服务器地址信息结构*/
Server_add.sin_family=AF_INET;             /*地址家族，必须是 AF_INET，注意只有它不是网络字节顺序*/
Server_add.sin_addr.S_un.S_addr=htonl(INADDR_ANY);             /*主机地址*/
```

```
Server_add.sin_port=htons(5000);                                   /*端口号*/
bind(socket_server,(SOCKADDR*)&Server_add,sizeof(SOCKADDR))        /*使用 bind 函数进行绑定*/
```

4．listen 函数

该函数的功能是将套接字设置为监听模式。对于流式套接字，必须处于监听模式才能够接收客户端套接字的连接。该函数的原型如下：

```
int listen(SOCKET s, int backlog);
```

- ☑ s：表示套接字标识。
- ☑ backlog：表示等待连接的最大队列长度。例如，如果 backlog 被设置为 2，此时有 3 个客户端同时发出连接请求，那么前两个客户端连接会放置在等待队列中，第 3 个客户端会得到错误信息。

例如，使用 listen 函数设置套接字为监听状态：

```
listen(socket_server,5);
```

设置套接字为监听状态，为连接做准备，最大等待的数目为 5。

5．accept 函数

该函数的功能是接受客户端的连接。在流式套接字中，只有在套接字处于监听状态，才能接受客户端的连接。该函数的原型如下：

```
SOCKET accept(SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen);
```

- ☑ s：是一个套接字，它应处于监听状态。
- ☑ addr：是一个 sockaddr_in 结构指针，包含一组客户端的端口号、IP 地址等信息。
- ☑ addrlen：用于接收参数 addr 的长度。
- ☑ 返回值：一个新的套接字，它对应于已经接受的客户端连接，对于该客户端的所有后续操作，都应使用这个新的套接字。

例如，使用 accept 函数接受客户端的连接请求：

```
/*接受客户端的发送请求，等待客户端发送 connect 请求*/
socket_receive=accept(socket_server,(SOCKADDR*)&Client_add,&Length);
```

其中，socket_receive 保存接受请求后返回的新的套接字，socket_server 为绑定在地址和端口上的套接字，而 Client_add 是有关客户端的 IP 地址和端口的信息结构，最后的 Length 是 Client_add 的大小。可以使用 sizeof 函数取得，然后用 Length 变量保存。

6．closesocket 函数

该函数的功能是关闭套接字。其原型如下：

```
int closesocket(SOCKET s);
```

其中，s 标识一个套接字。如果参数 s 设置了 SO_DONTLINGER 选项，则调用该函数后会立即返回。此时如果有数据尚未传送完毕，则会继续传递数据，然后再关闭套接字。

例如，使用 closesocket 函数关闭套接字，释放客户端的套接字资源。

```
closesocket(socket_receive);                                    /*释放客户端的套接字资源*/
```

在代码中，socket_receive 是一个套接字，当不使用时就可以利用 closesocket 函数将其资源释放。

7. connect 函数

该函数的功能是发送一个连接请求。其原型如下：

```
int connect(SOCKET s,const struct sockaddr FAR*    name,int namelen);
```

☑ s：表示一个套接字。

☑ name：表示套接字 s 要连接的主机地址和端口号。

☑ namelen：是 name 缓冲区的长度。

☑ 返回值：如果函数执行成功，则返回值为 0，否则为 SOCKET_ERROR。用户可以通过 WSAGETLASTERROR 得到其错误描述。

例如，使用 connect 函数与一个套接字建立连接：

```
connect(socket_send,(SOCKADDR*)&Server_add,sizeof(SOCKADDR));
```

在代码中，socket_send 表示要与服务器建立连接的套接字，而 Server_add 是要连接的服务器地址信息。

8. htons 函数

该函数的功能是将一个 16 位的无符号短整型数据由主机排列方式转换为网络排列方式。其原型如下：

```
u_short htons(u_short hostshort);
```

☑ hostshort：是一个主机排列方式的无符号短整型数据。

☑ 返回值：函数返回值是 16 位的网络排列方式数据。

例如，使用 htons 函数对一个无符号短整型数据进行转换：

```
Server_add.sin_port=htons(5000);
```

在代码中，Sever_add 是有关主机地址和端口的结构，其中 sin_port 表示的是端口号。因为端口号要使用网络排列方式，所以需要使用 htons 函数进行转换，从而设定端口号。

9. htonl 函数

该函数的功能是将一个无符号长整型数据由主机排列方式转换为网络排列方式。其原型如下：

```
u_long htonl(u_long hostlong);
```

☑ hostlong：表示一个主机排列方式的无符号长整型数据。

☑ 返回值：32 位的网络排列方式数据。

其使用方式与 htons 函数相似，不过是将一个 32 位数值转换为 TCP/IP 网络字节顺序。

10. inet_addr 函数

该函数的功能是将一个由字符串表示的地址转换为 32 位的无符号长整型数据。其原型如下：

```
unsigned long inet_addr(const char FAR * cp);
```

☑ cp：表示一个 IP 地址的字符串。
☑ 返回值：32 位无符号长整数。

例如，使用 inet_addr 函数将一个字符串转换成一个以点分十进制格式表示的 IP 地址（如 192.168.1.43）：

```
Server_add.sin_addr.S_un.S_addr = inet_addr("192.168.1.43");
```

在代码中设置服务器的 IP 地址为 198.168.1.43。

11. recv 函数

该函数的功能是从面向连接的套接字中接收数据。其原型如下：

```
int recv(SOCKET s,char FAR* buf,int len,int flags);
```

☑ s：表示一个套接字。
☑ buf：表示接收数据的缓冲区。
☑ len：表示 buf 的长度。
☑ flags：表示函数的调用方式。如果为 MSG_PEEK，则表示查看传来的数据，在序列前端的数据会被复制一份到返回缓冲区中，但是这个数据不会从序列中移走；如果为 MSG_OOB，则表示用来处理 Out-Of-Band 数据，也就是外带数据。

例如，使用 recv 函数接收数据：

```
recv(socket_send,Receivebuf,100,0);
```

其中，socket_send 是用于连接的套接字，而 Receivebuf 是用来接收保存数据的空间，100 是该空间的大小。

12. send 函数

该函数的功能是在面向连接方式的套接字间发送数据。其原型如下：

```
int send(SOCKET s,const char FAR * buf, int len,int flags);
```

☑ s：表示一个套接字。
☑ buf：表示存放要发送数据的缓冲区。
☑ len：表示缓冲区长度。
☑ flags：表示函数的调用方式。

例如，使用 send 函数发送数据：

```
send(socket_receive,Sendbuf,100,0);
```

在代码中，socket_receive 用于连接的套接字，而 Sendbuf 保存要发送的数据，100 为该数据的大小。

13．recvfrom 函数

该函数用于接收一个数据报信息并保存源地址。其原型如下：

```
int recvfrom(SOCKET s, char FAR* buf, int len, int flags, struct sockaddr FAR* from, int FAR* fromlen);
```

- ☑ s：表示准备接收数据的套接字。
- ☑ buf：指向缓冲区的指针，用来接收数据。
- ☑ len：表示缓冲区的长度。
- ☑ flags：通过设置该值可以影响函数的调用行为。
- ☑ from：是一个指向地址结构的指针，用来接收发送数据方的地址信息。
- ☑ fromlen：表示缓冲区的长度。

14．sendto 函数

该函数的功能是向一个特定的目的方发送数据。其原型如下：

```
int sendto(SOCKET s,const char FAR * buf,int len,int flags,const struct sockaddr FAR * to,int tolen);
```

- ☑ s：表示一个（可能已经建立连接的）套接字的标识符。
- ☑ buf：指向缓冲区的指针，该缓冲区包含将要发送的数据。
- ☑ len：表示缓冲区的长度。
- ☑ flags：通过设置该值可以影响函数的调用行为。
- ☑ to：指定目标套接字的地址。
- ☑ tolen：表示缓冲区的长度。

15．WSACleanup 函数

该函数的功能是释放为 Ws2_32.dll 动态链接库初始化时分配的资源。其原型如下：

```
int WSACleanup(void);
```

使用该函数可关闭动态链接库：

```
WSACleanup();                                          /*关闭动态链接库*/
```

16.3.2 基于 TCP 的网络聊天程序

根据上面对于网络的学习，本节将编写一个基于 TCP 网络通信的聊天程序，希望读者通过这个程序，可对前面的学习内容有一个更好的理解。

【例 16.1】 网络聊天服务器端的程序。(**实例位置：资源包\TM\sl\16\1**)

本实例是基于 TCP 的网络聊天程序，根据有关 TCP 的套接字 socket 编程中服务器的设计过程，编写下面的代码：

```
#include<stdio.h>
#include<winsock.h>                                    /*引入 winsock 头文件*/
```

```
int main()
{
    /*------------------------------------------------*/
    /*------------------定义变量------------------------*/
    /*------------------------------------------------*/
    char Sendbuf[100];                     /*发送数据的缓冲区*/
    char Receivebuf[100];                  /*接收数据的缓冲区*/
    int SendLen;                           /*发送数据的长度*/
    int ReceiveLen;                        /*接收数据的长度*/
    int Length;                            /*表示 SOCKADDR 的大小*/

    SOCKET socket_server;                  /*定义服务器套接字*/
    SOCKET socket_receive;                 /*定义用于连接套接字*/

    SOCKADDR_IN Server_add;                /*服务器地址信息结构*/
    SOCKADDR_IN Client_add;                /*客户端地址信息结构*/

    WORD wVersionRequested;                /*字（word）：unsigned short*/
    WSADATA wsaData;                       /*库版本信息结构*/
    int error;                             /*表示错误*/

    /*------------------------------------------------*/
    /*------------------初始化套接字库--------------------*/
    /*------------------------------------------------*/
    /*定义版本类型。将两个字节组合成一个字，前面是低字节，后面是高字节*/
    wVersionRequested = MAKEWORD(2, 2);
    /*加载套接字库，初始化 Ws2_32.dll 动态链接库*/
    error = WSAStartup(wVersionRequested, &wsaData);
    if(error!=0)
    {
        printf("加载套接字失败！");
        return 0;                          /*程序结束*/
    }
    /*判断请求加载的版本号是否符合要求*/
    if(LOBYTE(wsaData.wVersion) != 2 ||
            HIBYTE(wsaData.wVersion) != 2)
    {
        WSACleanup();                      /*不符合，关闭套接字库*/
        return 0;                          /*程序结束*/
    }

    /*------------------------------------------------*/
    /*----------------设置连接地址----------------------*/
    /*------------------------------------------------*/
    Server_add.sin_family=AF_INET;/*地址家族，必须是 AF_INET，注意只有它不是网络字节顺序*/
    Server_add.sin_addr.S_un.S_addr=htonl(INADDR_ANY);/*主机地址*/
    Server_add.sin_port=htons(5000);/*端口号*/

    /*---------------创建套接字-------------------------*/
```

```
/*AF_INET 表示指定地址族，SOCK_STREAM 表示流式套接字 TCP，特定的地址家族相关的协议*/
socket_server=socket(AF_INET,SOCK_STREAM,0);

/*--------------------------------------------------------*/
/*---绑定套接字到本地的某个地址和端口上---*/
/*--------------------------------------------------------*/
            /*socket_server 为套接字，(SOCKADDR*)&Server_add 为服务器地址*/
if(bind(socket_server,(SOCKADDR*)&Server_add,sizeof(SOCKADDR))==SOCKET_ERROR)
{
      printf("绑定失败\n");
}

/*--------------------------------------------------------*/
/*------------------设置套接字为监听状态-------------*/
/*--------------------------------------------------------*/
      /*监听状态，为连接做准备，最大等待的数目为 5*/
if(listen(socket_server,5)<0)
{
      printf("监听失败\n");
}

/*--------------------------------------------------------*/
/*--------------------接受连接-------------------------*/
/*--------------------------------------------------------*/
Length=sizeof(SOCKADDR);
/*接受客户端的发送请求，等待客户端发送 connect 请求*/
socket_receive=accept(socket_server,(SOCKADDR*)&Client_add,&Length);
if(socket_receive==SOCKET_ERROR)
{
      printf("接受连接失败");
}

/*--------------------------------------------------------*/
/*----------------进行聊天-------------------------*/
/*--------------------------------------------------------*/
while(1)                                         /*无限循环*/
{
      /*----------------接收数据--------------------------*/
      ReceiveLen =recv(socket_receive,Receivebuf,100,0);
      if(ReceiveLen<0)
      {
            printf("接收失败\n");
            printf("程序退出\n");
            break;
      }
      else
      {
            printf("client say: %s\n",Receivebuf);
      }
```

```
        /*----------------发送数据----------------------------*/
        printf("please enter message:");
        scanf("%s",Sendbuf);
        SendLen=send(socket_receive,Sendbuf,100,0);
        if(SendLen<0)
        {
            printf("发送失败\n");
        }
    }

    /*------------------------------------------------------*/
    /*-------------------释放套接字，关闭动态库-------------*/
    /*------------------------------------------------------*/
    closesocket(socket_receive);                /*释放客户端的套接字资源*/
    closesocket(socket_server);                 /*释放套接字资源*/
    WSACleanup();                               /*关闭动态链接库*/
    return 0;
}
```

在运行程序之前，要添加相应的库文件 ws2_32.lib，如图 16.2 所示。

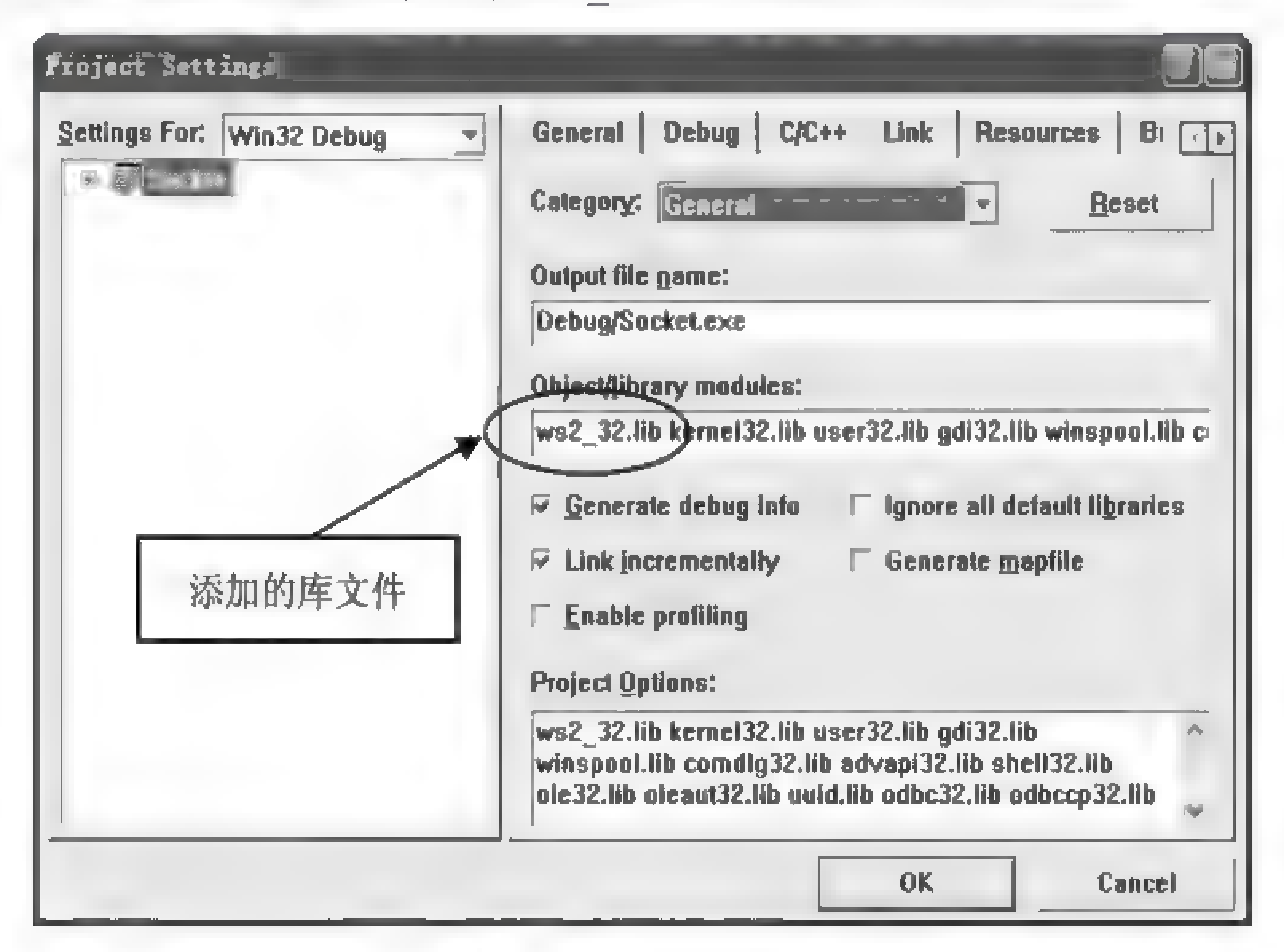


图 16.2　添加的库文件

以上就是有关服务器端的代码。整个程序流程按照以下顺序编写：

（1）创建套接字。

（2）绑定套接字到本地的地址和端口上。

（3）设置套接字为监听状态。

（4）接受请求连接的请求。

（5）进行通信。

（6）通信完毕，释放套接字资源。

【例 16.2】 网络聊天客户端的程序。（**实例位置：资源包\TM\sl\16\2**）

根据有关 TCP 的套接字 socket 编程中的客户端设计过程，编写下面的代码：

```
#include<stdio.h>
#include<winsock.h>                                    /*引入 winsock 头文件*/

int main()
{
    /*----------------------------------------------*/
    /*---------------定义变量--------------------*/
    /*----------------------------------------------*/
    char Sendbuf[100];                                 /*发送数据的缓冲区*/
    char Receivebuf[100];                              /*接收数据的缓冲区*/
    int SendLen;                                       /*发送数据的长度*/
    int ReceiveLen;                                    /*接收数据的长度*/

    SOCKET socket_send;                                /*定义套接字*/
    SOCKADDR_IN Server_add;                            /*服务器地址信息结构*/

    WORD wVersionRequested;                            /*字（word）：unsigned short*/
    WSADATA wsaData;                                   /*库版本信息结构*/
    int error;                                         /*表示错误*/

    /*----------------------------------------------*/
    /*---------------初始化套接字库---------------*/
    /*----------------------------------------------*/
                /*定义版本类型。将两个字节组合成一个字，前面是低字节，后面是高字节*/
    wVersionRequested = MAKEWORD(2, 2);
                /*加载套接字库，初始化 Ws2_32.dll 动态链接库*/
    error = WSAStartup(wVersionRequested, &wsaData);
    if(error!=0)
    {
        printf("加载套接字失败！");
        return 0;                                      /*程序结束*/
    }
                /*判断请求加载的版本号是否符合要求*/
    if(LOBYTE(wsaData.wVersion) != 2 ||
          HIBYTE(wsaData.wVersion) != 2)
    {
        WSACleanup();                                  /*不符合，关闭套接字库*/
        return 0;                                      /*程序结束*/
    }

    /*----------------------------------------------*/
    /*---------------设置服务器地址---------------*/
    /*----------------------------------------------*/
    Server_add.sin_family=AF_INET;/*地址家族，必须是 AF_INET，注意只有它不是网络字节顺序*/
    /*服务器的地址，将一个点分十进制表示为 IP 地址，inet_ntoa 是将地址转换成字符串*/
    Server_add.sin_addr.S_un.S_addr = inet_addr("192.168.1.43");
    Server_add.sin_port=htons(5000);/*端口号*/
```

```
/*------------------------------------------------------------*/
/*------------------------进行连接服务器---------------------*/
/*------------------------------------------------------------*/
/*客户端创建套接字，但是不需要绑定，只需要和服务器建立起连接即可*/
/*socket_sendr 表示的是套接字，Server_add 是服务器的地址结构*/
socket_send=socket(AF_INET,SOCK_STREAM,0);

/*------------------------------------------------------------*/
/*---------------------创建用于连接的套接字-----------------*/
/*------------------------------------------------------------*/
/*AF_INET 表示指定地址族，SOCK_STREAM 表示流式套接字 TCP，特定的地址家族相关的协议*/
if(connect(socket_send,(SOCKADDR*)&Server_add,sizeof(SOCKADDR)) == SOCKET_ERROR)
{
	printf("连接失败!\n");
}

/*------------------------------------------------------------*/
/*------------------------进行聊天-----------------------------*/
/*------------------------------------------------------------*/
while(1)                                             /*无限循环*/
{
	/*-------------发送数据过程-------------*/
	printf("please enter message:");
	scanf("%s",Sendbuf);
	SendLen = send(socket_send,Sendbuf,100,0);           /*发送数据*/
	if(SendLen < 0)
	{
		printf("发送失败!\n");
	}

	/*-------------接收数据过程-------------*/
	ReceiveLen =recv(socket_send,Receivebuf,100,0);      /*接收数据*/
	if(ReceiveLen<0)
	{
		printf("接收失败\n");
		printf("程序退出\n");
		break;                                           /*跳出循环*/
	}
	else
	{
		printf("Server say: %s\n",Receivebuf);
	}
}

/*------------------------------------------------------------*/
/*----------------------释放套接字，关闭动态库--------------*/
/*------------------------------------------------------------*/
closesocket(socket_send);                            /*释放套接字资源*/
WSACleanup();                                        /*关闭动态链接库*/
```

```
    return 0;
}
```

以上就是有关客户端的代码。整个程序流程按照以下顺序编写：

（1）创建套接字。

（2）发出连接请求。

（3）请求连接后进行通信操作。

（4）释放套接字资源。

先运行例 16.1，然后运行例 16.2。首先在客户端输入数据，当按 Enter 键后，即可以在服务器端看到输入的信息。当客户端输入完后，服务器端就可以对其进行回复，输入数据后按 Enter 键发送消息，将数据发送到客户端。

客户端程序运行效果如图 16.3 所示。服务器端程序运行效果如图 16.4 所示。



图 16.3　客户端

图 16.4　服务器端

16.4　小　　结

本章主要介绍了使用 Windows Sockets 编写网络应用程序方面的内容。首先讲解了计算机网络的基本知识，引出了 Socket 套接字被引入到编写网络程序中的过程。然后讲解了套接字，其中包含两种基于 TCP 和 UDP 使用套接字编写网络应用程序的简单流程；接着讲解了编写网络应用程序要使用的一些基本函数；最后通过一个基于 TCP 的网络聊天程序，对本章所讲解的知识进行了一下整体的应用。希望读者仿照这个实例自己动手编写一个程序，这样对网络程序会有更好的理解。

16.5　实践与练习

1．设计程序，要求当客户端连接到服务器一端时，服务器会显示连接的提示信息，并反馈信息给客户。（**答案位置：资源包\TM\sl\16\3**）

2．修改例 16.1 和例 16.2，使其程序为基于 UCP 的网络聊天程序。（**答案位置：资源包\TM\ sl\16\4**）



项目实战

本篇通过一个大型的学生成绩管理系统，运用软件工程的设计思想，讲解如何进行软件项目的开发。书中按照“编写需求分析→系统设计→功能设计→创建项目→实现项目模块功能→运行项目”的步骤，带领读者一步一步地亲身体验项目开发的全过程。

第17章

学生成绩管理系统

（视频讲解：40分钟）

通过前面章节的学习，读者应该对C语言的基本概念和知识体系有了一定的了解，本章将在前面学习的基础上设计一个学生成绩管理系统，来对前面学过的知识加以巩固。本实例将综合应用前面学过的很多内容，并详细介绍该程序的开发过程。

通过阅读本章，您可以：

- 掌握如何进行需求分析
- 掌握如何进行系统设计
- 掌握功能设计中各个模块的设计方法



17.1　需求分析

目前，各类学校的在校生人数都在不断增加，而且不同专业的学生选修课、实验课、考试课分别占的比重不同，依靠传统的方式管理学生成绩信息，会给日常的管理工作带来诸多不便。计算机信息技术的发展，为学生成绩管理注入了新的生机。通过对市场的调查，一个合格的学生成绩管理系统必须具备以下 4 种功能：

☑　能够对学生成绩信息进行集中管理。

☑　能够大大提高用户的工作效率。

☑　能够对学生成绩信息实现增、删、改。

☑　能够按成绩信息进行排序。

一个学生成绩管理系统最重要的功能包括学生成绩的添加、删除、查询、修改、指定位置插入及排名。其中，学生成绩信息的查询、删除、修改、指定位置的插入等都要依靠输入的学生学号来实现，学生成绩排序是根据学生成绩由高到低进行排序的。

视频讲解

17.2　系统设计

根据上面的需求分析，得出该学生成绩管理系统要实现的功能有以下方面：

☑　录入学生成绩信息。

☑　实现删除功能，即输入学号，删除相应的记录。

☑　实现查找功能，即输入学号，查询该学生成绩的相关信息。

☑　实现修改功能，即输入学号，修改相应信息。

☑　指定位置插入学生成绩信息，即输入要插入的位置，并将新的信息插入指定位置。

☑　学生成绩排名，即按照总成绩进行由高到低的排名。

☑　统计保存的学生人数。

该学生成绩管理系统的结构设计如图 17.1 所示。

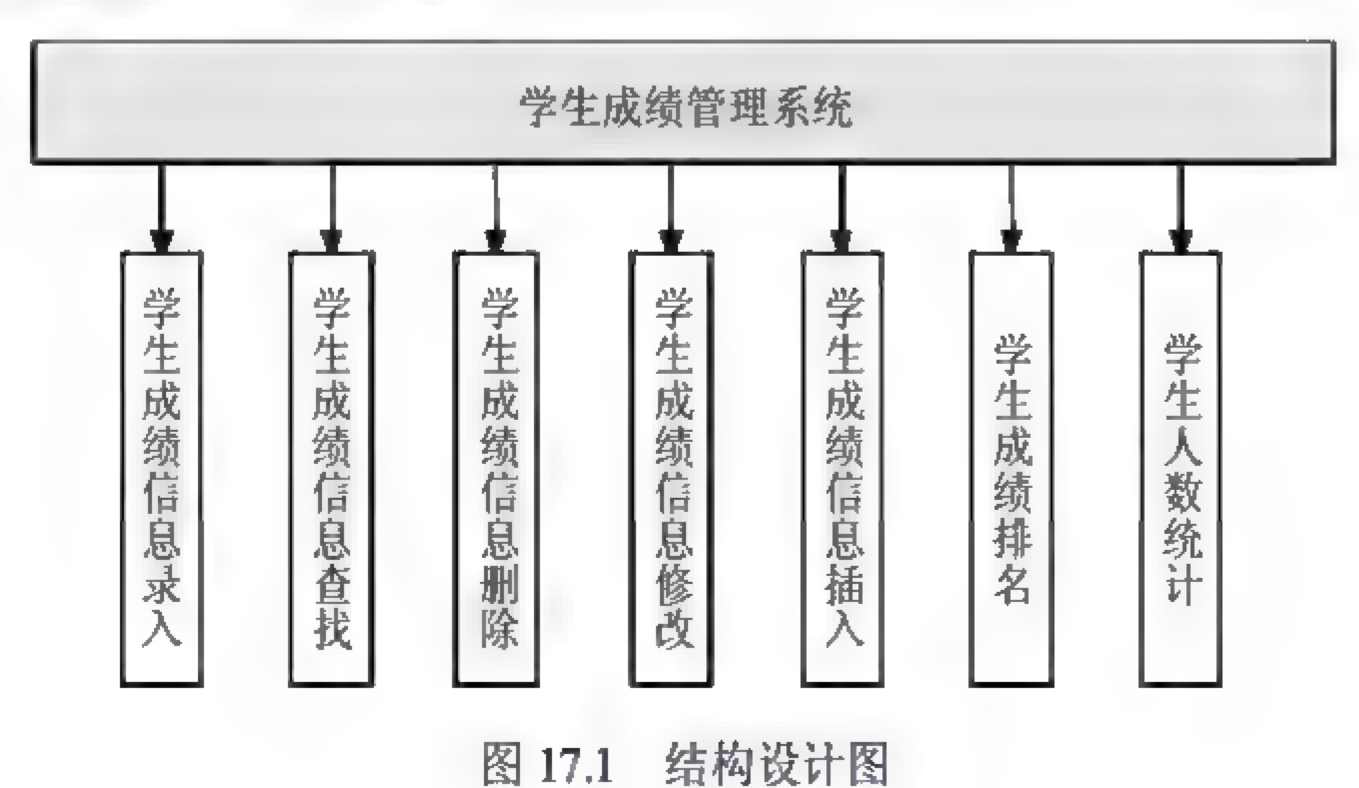


图 17.1　结构设计图



17.3 功 能 设 计

针对系统分析中提出的功能，介绍学生成绩的添加、删除、查询、修改、指定位置插入及排名程序的具体实现方法。

17.3.1 功能选择界面

功能选择界面会将该系统中的所有功能显示出来，每种功能前有对应的数字，输入对应的数字，即可选择相应的功能。程序运行结果如图 17.2 所示。



图 17.2　功能选择界面

程序代码如下：

```
void menu()/*自定义函数实现菜单功能*/
{
    system("cls");
    printf("\n\n\n\n\n");
    printf("\t\t|------------------STUDENT----------------|\n");
    printf("\t\t|\t 0. exit                               |\n");
    printf("\t\t|\t 1. input record                       |\n");
    printf("\t\t|\t 2. search record                      |\n");
    printf("\t\t|\t 3. delete record                      |\n");
    printf("\t\t|\t 4. modify record                      |\n");
    printf("\t\t|\t 5. insert record                      |\n");
    printf("\t\t|\t 6. order                              |\n");
    printf("\t\t|\t 7. number                             |\n");
    printf("\t\t|-----------------------------------------|\n\n");
    printf("\t\t\tchoose(0-7):");
}
```

menu 函数将程序中的基本功能列出。当输入相应数字后，程序会根据该数字调用不同的函数，当输入“0”时，退出该系统。这部分主要通过 main 函数实现，代码如下：

```
void main()/*主函数*/
{
	int n;
	menu();
	scanf("%d",&n);                                        /*输入选择功能的编号*/
	while(n)
	{
		switch(n)
		{
		case 1:
			in();
			break;
		case 2:
			search();
			break;
		case 3:
			del();
			break;
		case 4:
			modify();
			break;
		case 5:
			insert();
			break;
		case 6:
			order();
			break;
		case 7:
			total();
			break;
		default:break;
		}
		getch();
		menu();                                        /*执行完功能后再次显示菜单界面*/
		scanf("%d",&n);
	}
}
```

在 main 函数中分别调用了 in、search、del、modify、insert、order、total 等函数，这些函数实现的功能将在下面的内容中进行详细介绍。

17.3.2　录入学生成绩信息

当输入“1”时，进入学生成绩信息录入界面，如图 17.3 所示。

```
number  name          elective    experiment   required     sum
21      nini          85.0        90.0         86.0         86.7
22      kiki          95.0        96.0         98.0         97.3
please input(y/n):y
please input per centum:
elective:0.1
experiment:0.2
required course:0.7
number:23
name:gigi
elective:82
experiment:81
required course:86
gigi saved!
continue?(y/n):
```

图 17.3　学生成绩信息录入界面

从图 17.3 中可以发现，在录入新的信息之前会将原有的记录显示出来。当无记录时，会提示“No record”；当要输入信息时，输入“y”或“Y”，然后按照给出的提示信息输入即可；当要退出该功能时，输入除“y”和“Y”之外的任意键即可。主要程序代码如下：

```
void in()                                           /*录入学生信息*/
{
int i,m=0;                                          /*m 是记录的条数*/
char ch[2];
FILE *fp;                                           /*定义文件指针*/
if((fp=fopen("data","ab+"))==NULL)                  /*打开指定文件*/
{
    printf("can not open\n");
    return;
}
while(!feof(fp))
{
    if(fread(&stu[m] ,LEN,1,fp)==1)
        m++;                                        /*统计当前记录条数*/
}
fclose(fp);
if(m==0)
printf("No record!\n");
else
{
    system("cls");
    show();                                         /*调用 show 函数，显示原有信息*/
}
if((fp=fopen("data","wb"))==NULL)
{
    printf("can not open\n");
    return;
}
for(i=0;i<m;i++)
```

```
fwrite(&stu[i] ,LEN,1,fp);                          /*向指定的磁盘文件写入信息*/
printf("please input(y/n):");
scanf("%s",ch);
if(strcmp(ch,"Y")==0||strcmp(ch,"y")==0)
{
printf("please input per centum:");
printf("\nelective:");
scanf("%f",&Felect);
printf("\nexperiment:");
scanf("%f",&Fexpe);
printf("\nrequired course:");
scanf("%f",&Frequ);
}
while(strcmp(ch,"Y")==0||strcmp(ch,"y")==0)         /*判断是否要录入新信息*/
{
    printf("number:");
    scanf("%d",&stu[m].num);                        /*输入学生学号*/
    for(i=0;i<m;i++)
        if(stu[i].num==stu[m].num)
        {
            printf("the number is existing,press any to continue!");
            getch();
            fclose(fp);
            return;
        }
        printf("name:");
        scanf("%s",stu[m].name);                    /*输入学生姓名*/
        printf("elective:");
        scanf("%lf",&stu[m].elec);                  /*输入选修课成绩*/
        printf("experiment:");
        scanf("%lf",&stu[m].expe);                  /*输入实验课成绩*/
        printf("required course:");
        scanf("%lf",&stu[m].requ);                  /*输入必修课成绩*/
        stu[m].sum=stu[m].elec*Felect+stu[m]. Fexpe*lexpe+stu[m].requ*Frequ;     /*计算出总成绩*/
        if(fwrite(&stu[m],LEN,1,fp)!=1)             /*将新录入的信息写入指定的磁盘文件*/
        {
            printf("can not save!");
            getch();
        }
        else
        {
            printf("%s saved!\n",stu[m].name);
            m++;
        }
        printf("continue?(y/n):");                  /*询问是否继续*/
        scanf("%s",ch);
}
fclose(fp);
printf("OK!\n");
}
```

17.3.3 查询学生成绩信息

查询学生成绩信息只需要输入学号便可进行。若该学号存在，则会提示是否显示该条信息；若不存在，则会输出提示信息。查询界面如图 17.4 所示。



图 17.4 查询界面

实现上述功能的程序代码如下：

```
void search()                                          /*自定义查找函数*/
{
    FILE *fp;
    int snum,i,m=0;
    char ch[2];
    if((fp=fopen("data","ab+"))==NULL)
    {
        printf("can not open\n");
        return;
    }
    while(!feof(fp))
        if(fread(&stu[m],LEN,1,fp)==1)
            m++;
        fclose(fp);
        if(m==0)
        {
            printf("no record!\n");
            return;
        }
        printf("please input the number:");
        scanf("%d",&snum);
        for(i=0;i<m;i++)
            if(snum==stu[i].num)                       /*查找输入的学号是否在记录中*/
            {
                printf("find the student,show?(y/n)");
                scanf("%s",ch);
                if(strcmp(ch,"Y")==0||strcmp(ch,"y")==0)
                {
                    printf("number name elective experiment required sum\t\n");
                    printf(FORMAT,DATA);               /*将查找出的结果按指定格式输出*/
                    break;
```

```
                    }
                else
                    return;
            }
            if(i==m)
                printf("can not find the student!\n");   /*未找到要查找的信息*/
}
```

17.3.4　删除学生成绩信息

输入要删除的学生的学号，如果该学号存在，则提示是否删除；若不存在，则给出提示信息。删除学生成绩信息的界面如图 17.5 所示。



图 17.5　删除学生成绩信息界面

实现上述功能的程序代码如下：

```
void del()                                              /*自定义删除函数*/
{
    FILE *fp;
    int snum,i,j,m=0;
    char ch[2];
    if((fp=fopen("data","ab+"))==NULL)
    {
        printf("can not open\n");
        return;
    }
    while(!feof(fp))
        if(fread(&stu[m],LEN,1,fp)==1)
            m++;
        fclose(fp);
        if(m==0)
        {
            printf("no record!\n");
            return;
        }
        printf("please input the number:");
        scanf("%d",&snum);
        for(i=0;i<m;i++)
            if(snum==stu[i].num)
                break;
```

```
            printf("find the student,delete?(y/n)");
            scanf("%s",ch);
            if(strcmp(ch,"Y")==0||strcmp(ch,"y")==0)        /*判断是否要进行删除*/
            {
                for(j=i;j<m;j++)
                    stu[j]=stu[j+1];                        /*将后一个记录移到前一个记录的位置*/
                m--;                                        /*记录的总个数减 1*/
                  printf("delete successfully!\n");
            }
                if((fp=fopen("data","wb"))==NULL)
                {
                    printf("can not open\n");
                    return;
                }
                for(j=0;j<m;j++)                            /*将更改后的记录重新写入指定的磁盘文件中*/
                    if(fwrite(&stu[j] ,LEN,1,fp)!=1)
                    {
                        printf("can not save!\n");
                        getch();
                    }
                    fclose(fp);
                    printf("delete successfully!\n");
}
```

17.3.5 修改学生成绩信息

输入学号，若该学号存在，则修改该学号所对应的学生成绩信息，并保存；若不存在，则给出相应的提示信息。运行界面如图 17.6 所示。

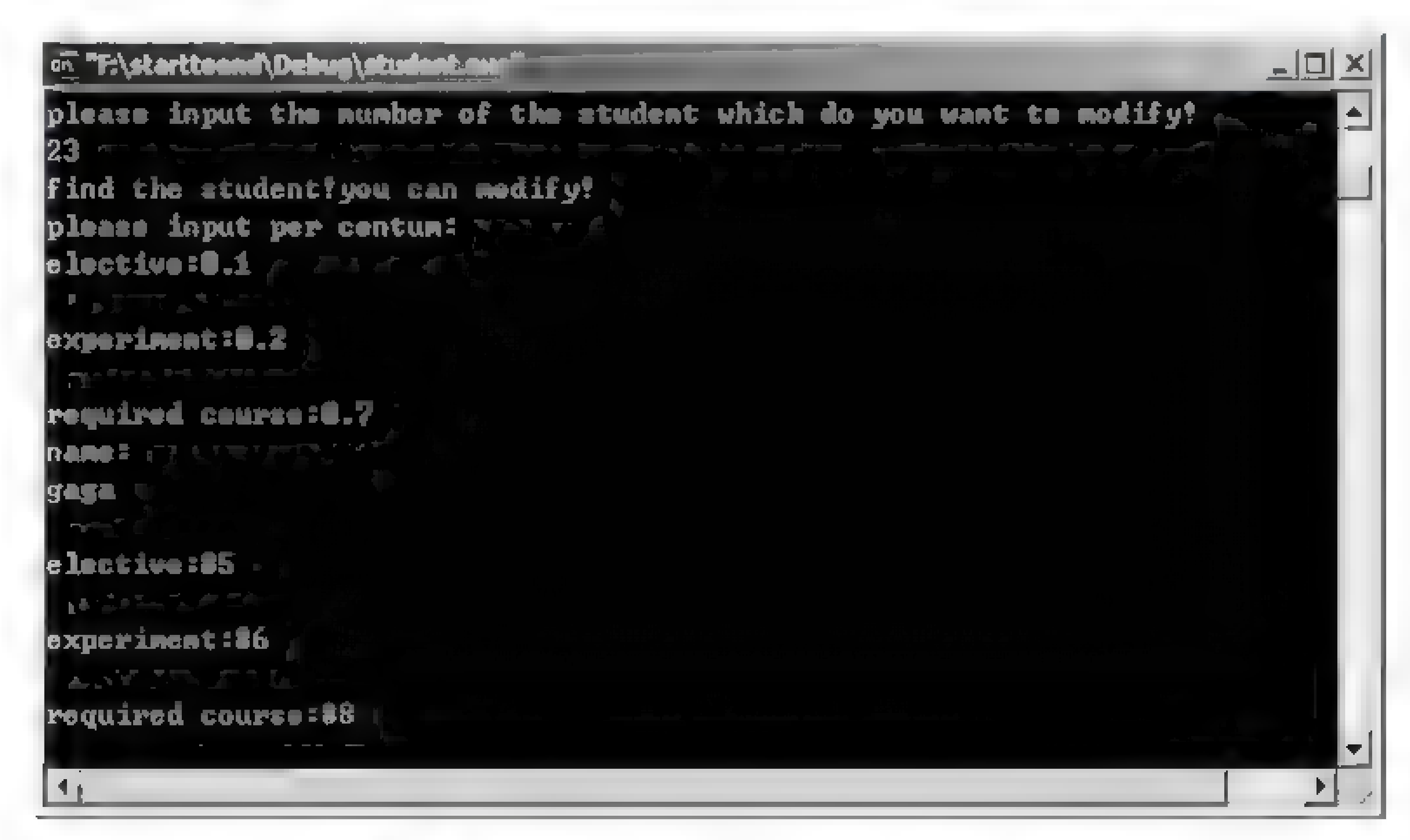


图 17.6　修改学生成绩信息界面

实现上述功能的程序代码如下：

```
void modify()                                                   /*自定义修改函数*/
{
```

```
FILE *fp;
int i,j,m=0,snum;
if((fp=fopen("data","ab+"))==NULL)
{
	printf("can not open\n");
	return;
}
while(!feof(fp))
	if(fread(&stu[m],LEN,1,fp)==1)
		m++;
	if(m==0)
	{
		printf("no record!\n");
		fclose(fp);
		return;
	}
		printf("please input the number of the student which do you want to modify!\n");
	scanf("%d",&snum);
	for(i=0;i<m;i++)
		if(snum==stu[i].num)                       /*检索记录中是否有要修改的信息*/
			break;
		if(i<m)
		{
			printf("find the student!you can modify!\n");
			printf("please input per centum:");
			printf("\nelective:");
			scanf("%f",&Felect);
			printf("\nexperiment:");
			scanf("%f",&Fexpe);
			printf("\nrequired course:");
			scanf("%f",&Frequ);
			printf("name:\n");
			scanf("%s",stu[i].name);               /*输入名字*/
			printf("\nelective:");
			scanf("%lf",&stu[i].elec);             /*输入选修课成绩*/
			printf("\nexperiment:");
			scanf("%lf",&stu[i].expe);             /*输入实验课成绩*/
			printf("\nrequired course:");
			scanf("%lf",&stu[i].requ);             /*输入必修课成绩*/
			stu[i].sum=stu[i].elec*Felect+stu[i].expe*Fexpe+stu[i].requ*Frequ;
		}
		else
		{
			printf("can not find!");
			getchar();
			return;
		}
		if((fp=fopen("data","wb"))==NULL)
		{
			printf("can not open\n");
```

```
                return;
            }
            for(j=0;j<m;j++)                                    /*将新修改的信息写入指定的磁盘文件中*/
                if(fwrite(&stu[j] ,LEN,1,fp)!=1)
                {
                    printf("can not save!");
                    getch();
                }
                fclose(fp);
}
```

17.3.6 插入学生成绩信息

先输入学号，该学号用于确定要插入的位置，然后将新信息插入该学号之后。若输入的学号存在，则在输出提示信息后按任意键，返回操作界面；若该学号不存在，则可进行正常的信息录入。插入学生成绩信息的操作界面如图 17.7 所示。

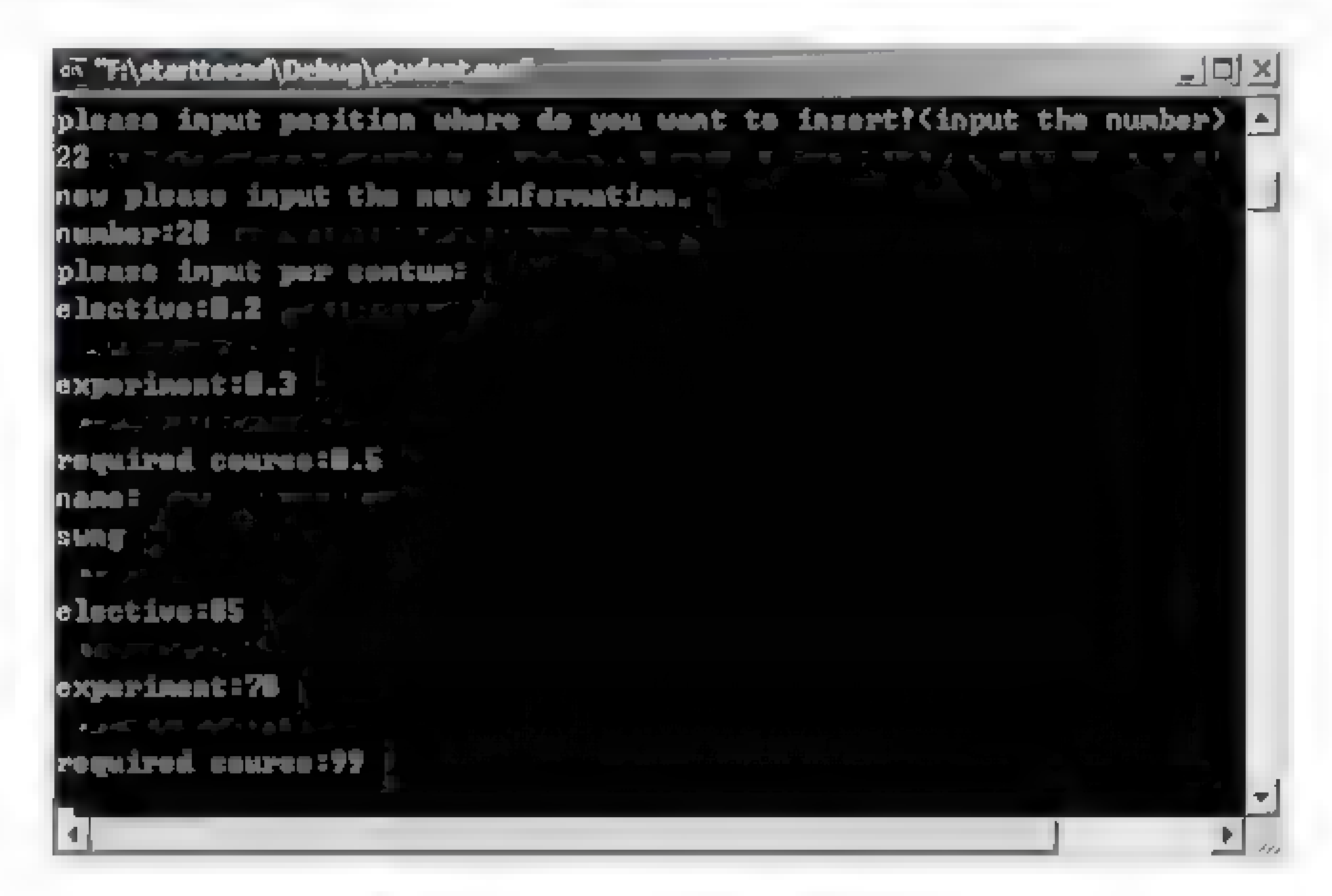


图 17.7　插入学生成绩信息界面

实现上述功能的程序代码如下：

```
void insert()                                                   /*自定义插入函数*/
{
    FILE *fp;
    int i,j,k,m=0,snum;
    if((fp=fopen("data","ab+"))==NULL)
    {
        printf("can not open\n");
        return;
    }
    while(!feof(fp))
        if(fread(&stu[m],LEN,1,fp)==1)
            m++;
        if(m==0)
        {
            printf("no record!\n");
```

```
            fclose(fp);
            return;
        }
        printf("please input position where do you want to insert!(input the number)\n");
        scanf("%d",&snum);                                  /*输入要插入的位置*/
        for(i=0;i<m;i++)
            if(snum==stu[i].num)
                break;
            for(j=m-1;j>i;j--)
                stu[j+1]=stu[j];                            /*从最后一条记录开始，均向后移一位*/
            printf("now please input the new information.\n");
            printf("number:");
            scanf("%d",&stu[i+1].num);
            for(k=0;k<m;k++)
                if(stu[k].num==stu[i+1].num&&k!=i+1)        /*查找要插入的位置*/
                {
                    printf("the number is existing,press any to continue!");
                    getch();
                    fclose(fp);
                    return;
                }
                printf("please input per centum:");         /*提示输入百分比*/
                printf("\nelective:");
                scanf("%f",&Felec);
                printf("\nexperiment:");
                scanf("%f",&Fexpe);
                printf("\nrequired course:");
                scanf("%f",&Frequ);
                printf("name:\n");
                scanf("%s",stu[i+1].name);                  /*输入学生姓名*/
                printf("\nelective:");
                scanf("%lf",&stu[i+1].elec);
                printf("\nexperiment:");
                scanf("%lf",&stu[i+1].expe);
                printf("\nrequired course:");
                scanf("%lf",&stu[i+1].requ);
                stu[i+1].sum=stu[i+1].elec*Felec+stu[i+1].expe*Fexpe+stu[i+1].requ*Frequ; /*计算总成绩*/
                if((fp=fopen("data","wb"))==NULL)
                {
                    printf("can not open\n");
                    return;
                }
                for(k=0;k<=m;k++)
                    if(fwrite(&stu[k] ,LEN,1,fp)!=1)        /*将修改后的记录写入磁盘文件中*/
                    {
                        printf("can not save!");
                        getch();
                    }
                    fclose(fp);
}
```

17.3.7 统计学生人数

如果选择 number 功能，即统计学生人数（信息条数），就会出现如图 17.8 所示的信息。



图 17.8 统计学生人数

实现上述功能的程序代码如下：

```
void total()
{
	FILE *fp;
	int m=0;
	if((fp=fopen("data","ab+"))==NULL)			/*判断文件是否打开成功*/
	{
		printf("can not open\n");
		return;
	}
	while(!feof(fp))
		if(fread(&stu[m],LEN,1,fp)==1)
			m++;						/*统计记录个数，即学生个数*/
		if(m==0)
		{
			printf("no record!\n");
			fclose(fp);
			return;
		}
		printf("the class are %d students!\n",m);		/*将统计的个数输出*/
		fclose(fp);
}
```

17.4 小 结

本章通过学生成绩管理系统的开发，介绍了开发 C 语言系统的流程和技巧。本实例并没有太多难点，例中介绍的几种功能都是在对文件进行操作的基础上实现的。通过该实例的学习，读者可以体验一下一个管理系统开发的全过程，为今后开发其他系统程序奠定基础。只要读者能够多读、多写、多练习，那么编写程序根本不是一个很难的过程。

附录 ASCII 表

ASCII 值	缩写/字符	解　释
0	NUL（null）	空字符（\0）
1	SOH（star to fhanding）	标题开始
2	STX（star to ftext）	正文开始
3	ETX（end of text）	正文结束
4	EOT（end of transmission）	传输结束
5	ENQ（enquiry）	请求
6	ACK（acknowledge）	收到通知
7	BEL（bell）	响铃（\a）
8	BS（backspace）	退格（\b）
9	HT（horizontal tab）	水平制表符（\t）
10	LF（NL）（linefeed,newline)	换行键（\n）
11	VT（verticaltab）	垂直制表符
12	FF（NP）（formfeed,newpage）	换页键（\f）
13	CR（carriagereturn）	回车键（\r）
14	SO（shift out）	不用切换
15	SI（shift in）	启用切换
16	DLE（data link escape）	数据链路转义
17	DC1（device control1）	设备控制 1
18	DC2（device control2）	设备控制 2
19	DC3（device control3）	设备控制 3
20	DC4（device control4）	设备控制 4
21	NAK（negative acknowledge）	拒绝接收
22	SYN（synchronousidle）	同步空闲
23	ETB（end of trans.block）	传输块结束
24	CAN（cancel）	取消
25	EM（end of medium）	介质中断
26	SUB（substitute）	替补
27	ESC（escape）	溢出
28	FS（file separator）	文件分割符
29	GS（group separator）	分组符
30	RS（record separator）	记录分离符
31	US（unit separator）	单元分隔符
32	SP（space）	
33	!	

续表

ASCII 值	缩写/字符	解　释
34	"	
35	#	
36	$	
37	%	
38	&	
39	’	
40	(	
41	)	
42	*	
43	+	
44	,	
45	-	
46	.	
47	/	
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	

续表

ASCII 值	缩写/字符	解　　释
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[	
92	\	
93	]	
94	^	
95	_	
96	`	
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	

续表

ASCII 值	缩写/字符	解　　释
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	
124	|	
125	}	
126	～	
127	DEL（Delete）	